



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)
- [Academic](#)

SQL in a Nutshell, 2nd Edition

By [Kevin E. Kline](#)

Publisher: O'Reilly

Pub Date: September 2004

ISBN: 0-596-00481-8

Pages: 710

This essential desktop reference drills down and documents every SQL command and how to use it in both commercial (Oracle, DB2, and Microsoft SQL Server) and open source implementations (PostgreSQL, and MySQL). It includes the command syntax (by vendor, if the syntax differs across implementations), a description, and practical examples. And it also explains how the leading commercial and open sources database product implement SQL. This wealth of information is packed into a succinct, comprehensive, and extraordinarily easy-to-use format.



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)
- [Academic](#)

SQL in a Nutshell, 2nd Edition

By [Kevin E. Kline](#)

Publisher: O'Reilly

Pub Date: September 2004

ISBN: 0-596-00481-8

Pages: 710

[Copyright](#)

[Preface](#)

[Why This Book?](#)

[Who Should Read This Book?](#)

[How This Book Is Organized](#)

[Conventions Used in This Book](#)

[How to Use This Book](#)

[How to Contact Us](#)

[Safari Enabled](#)

[Resources](#)

[Changes in the Second Edition](#)

[Acknowledgments](#)

[Chapter 1. SQL History and Implementations](#)

[Section 1.1. The Relational Model and ANSI SQL](#)

[Section 1.2. History of the SQL Standard](#)

[Section 1.3. SQL Dialects](#)

[Chapter 2. Foundational Concepts](#)

[Section 2.1. Database Platforms Described in This Book](#)

[Section 2.2. Categories of Syntax](#)

[Section 2.3. SQL2003 and Platform-Specific Datatypes](#)

[Section 2.4. Constraints](#)

[Chapter 3. SQL Statement Command Reference](#)

[Section 3.1. How to Use This Chapter](#)

[Section 3.2. SQL Platform Support](#)

[Section 3.3. SQL Command Reference](#)

[Chapter 4. SQL Functions](#)

[Section 4.1. Types of Functions](#)

[Section 4.2. ANSI SQL Aggregate Functions](#)

[Section 4.3. ANSI SQL Window Functions](#)

[Section 4.4. ANSI SQL Scalar Functions](#)

[Section 4.5. Platform-Specific Extensions](#)

[Chapter 5. Database Programming](#)

[Section 5.1. Database Programming Overview](#)

[Section 5.2. Opening a Database Connection](#)

[Section 5.3. Closing a Database Connection](#)

[Section 5.4. Managing Transactions](#)

[Section 5.5. Executing Statements](#)

[Section 5.6. Retrieving Data](#)

[Section 5.7. Bound Parameters](#)

[Section 5.8. Error Handling](#)

[Section 5.9. Examples](#)

[Appendix A. Sybase Adaptive Server](#)

[Section A.1. Sybase Adaptive Server Datatypes](#)

[Section A.2. Sybase Adaptive Server SQL Statements](#)

[Section A.3. Sybase Adaptive Server SQL Functions](#)

[Section A.4. Sybase Adaptive Server Keywords](#)

[Appendix B. Shared and Platform-Specific Keywords](#)

[Colophon](#)

[Index](#)



Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

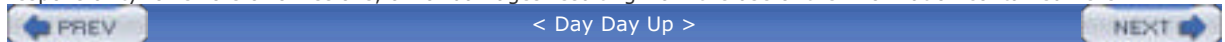
Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The In a Nutshell series designations, *SQL in a Nutshell*, the image of a chameleon, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



Preface

Since its first incarnation in the 1970s, the Structured Query Language (SQL) has been developed hand in hand with the information boom, and as a result, is the most widely used database manipulation language in business and industry. A number of different software companies and program developers, including those in the open source movement, have concurrently developed their own SQL dialects in response to specific professional needs. All the while, standards bodies have developed a growing list of common features.

SQL in a Nutshell, Second Edition, describes the latest ANSI standard, SQL2003, version of each SQL command, and then documents each platform's implementation of that command. In this book, you will find a concise explanation of the Relational Database Management System (RDBMS) model, a clear-cut explanation of foundational RDBMS concepts, and thorough coverage of SQL syntax and commands.

Most importantly, at least if you're a programmer or developer, is that *SQL in a Nutshell*, Second Edition, provides a concise guide both to the most popular commercial database packages on the market (Microsoft SQL Server, IBM's DB2 Universal Database, Sybase Adaptive Server, and Oracle), and to two of the best-known open source (<http://www.opensource.org>) database products (MySQL and PostgreSQL). *SQL in a Nutshell's* attention to open source SQL platforms is an affirmation of the growing importance of the open source movement within the computing community.

The database platforms covered in this book include:

- IBM DB2 UDB Version 8.0 for Linux, Unix, and Windows
- MySQL Version 4
- Oracle Database 10g
- PostgreSQL Version 7.0
- Microsoft SQL Server 2000
- Sybase Adaptive Server Version 12.5

Why This Book?

The primary source of information for relational databases is the documentation and help files provided by the vendors themselves. While each vendor's documentation is an indispensable resource that most database programmers and database administrators turn to first, it has a number of limitations:

- It covers only a single, specific vendor's product. There is no coverage for translation, migration, or integration issues.
- It describes programming methods, usually in a multitude of small, disconnected documents or help files.
- It covers individual commands, often in confusing detail, obscuring the simple and direct uses of commands that are used every day by programmers and administrators.

In other words, the documentation included with the vendor database is an exhaustive explanation of every aspect of their platform. After all, help texts are naturally geared toward delivering the main facts about the product. They'll tell you a command's specific syntax (and all its obscure variants) and, in general terms, how to implement it. However, if you move between RDMS's and you need to be productive very quickly, you will rarely use those obscure command variations and instead utilize the common capabilities in real-life situations.

This book begins where the vendor documentation ends by distilling the experiences of professional database administrators and developers who have used these SQL variants day in and day out to support complex enterprise applications. You'll get the benefit of their experience in a compact and easily usable format. Whether SQL is new to you or you have been using SQL since its earliest days, there are always new tips and techniques to learn. And, when moving between different implementations, it's always important to find out about the implementations that can bite you if you're not careful and informed.

Who Should Read This Book?

SQL in a Nutshell, Second Edition, benefits several groups of users. It is useful for programmers who require a concise and handy SQL reference tool. It is useful for developers who need to migrate from one SQL dialect to another. Finally, it is useful for database administrators (DBAs) who need to both execute a myriad of SQL statements to keep their enterprise databases up and running, and create and manage objects such as tables, indexes, and views.

This book is a reference work, not a tutorial. The writing is not expository. For example, we won't explain the concept of an elementary loop. Experienced developers already know such things. You want the meat. So we will explain, for example, the detailed workings of an ANSI standard cursor, how it works on each of the database platforms we cover, the special capabilities of cursors on each database platform, and the various pitfalls of cursors and how to get around them.

While we don't intend for *SQL in a Nutshell, Second Edition*, to be a tutorial on SQL or a handbook for database design, we do provide some brief coverage of those topics, and we hope you find that helpful. [Chapter 1](#) and [Chapter 2](#) provide a concise introduction to SQL, covering the general origins, essential structure, and basic operation of the language. If you're new to SQL, the introductions in [Chapter 1](#) and [Chapter 2](#) will help you to get started.

How This Book Is Organized

SQL in a Nutshell, Second Edition, is divided into five chapters and two appendixes:

[Chapter 1](#), *SQL History and Implementations*

Discusses the Relational Database Model, describes the current and previous SQL standards, and introduces the SQL implementations covered in this book.

[Chapter 2](#), *Foundational Concepts*

Describes the fundamental concepts necessary for understanding relational databases and SQL commands.

[Chapter 3](#), *SQL Statement Command Reference*

Provides an alphabetical command reference to SQL statements. It details the latest ANSI standard, SQL2003, for each command, as well as the implementation of each command by DB2, MySQL, Oracle, PostgreSQL, and SQL Server.

[Chapter 4](#), *SQL Functions*

Provides an alphabetical reference of the SQL2003 functions, describing vendor implementations of all SQL2003 functions. In addition, [Chapter 4](#) includes coverage for all platform-specific functions that are unique to each implementation.

[Chapter 5](#), *Database Programming*

Provides an overview of database programming from a variety of connectivity methods. ADO.NET and JDBC are discussed, among other topics.

[Appendix A](#), *Sybase Adaptive Server*

Sybase's Adaptive Server product and Microsoft SQL Server share a common heritage. As a result, most of the SQL Server commands detailed in [Chapter 3](#), and most of the functions detailed in [Chapter 4](#), will work with Sybase. However, "most" is the operative word, and we provide this appendix to show where Sybase differs from SQL Server.

[Appendix B](#), *Shared and Platform-Specific Keywords*

Provides a table of keywords declared in SQL2003 and by the different database platforms. You can use this table to look for words that you should not use for object or variable names.

Conventions Used in This Book

Constant Width

Used to indicate programming syntax, code fragments, and examples.

Constant Width Italic

Used to indicate variables in code that should be replaced with values.

Constant Width Bold

Used in code sections to highlight portions of the code.

Italic

Used to introduce new terms, for emphasis, to indicate commands or user-specified file and directory names, and to indicate variables within text.

Bold

Used to display the names of database objects, such as tables, columns, and stored procedures.

UPPERCASE ITALIC

Used to indicate SQL keywords when they appear in the text.



This icon indicates a tip, suggestion, or general note.



Indicates a warning or caution.

How to Use This Book

SQL in a Nutshell, Second Edition, is primarily a command reference. As a consequence, you'll probably use it to look up a variety of SQL commands and functions. However, with documentation for the ANSI standard itself, plus five database platforms, each command has the potential to get very large.

In order to reduce the verbiage describing each command, we compare each platform's implementation to the SQL2003 standard. If the platform supports a clause described in the SQL2003 discussion, then we won't repeat that clause again.

Generic and transportable examples are provided within the body of each SQL2003 command description. Since the SQL2003 standard is ahead of most database platforms, examples aren't provided for elements of the SQL2003 commands that are not supported by any platform discussed in this book. In addition, more examples are provided for each database platform that highlight unique extensions and enhancements.

We recognize that our approach may necessitate jumping from a description of a platform's implementation of a command back to the corresponding SQL2003 command description. However, we felt that this was better than packing the book with hundreds of pages of redundant content.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). We want to hear from you, especially with information that will make this book better. Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

We have a web site for the book, where we'll list any examples, errata, or plans for future editions. You can access this page at:

<http://www.oreilly.com/catalog/sqlnut2/>

Please help us out by pointing out any typos or syntactical errors that you encounter. (You can imagine how hard it is to proofread a book covering the ANSI standard and five separate products.) You may also ask technical questions or comment on the book by sending an email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly web site:

<http://www.oreilly.com>

Safari Enabled



When you see the Safari® Enabled icon on the back cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information.

Try it for free at <http://safari.oreilly.com>.

Resources

The following web sites provide additional information about the various platforms covered in this book:

DB2

IBM's DB2 database is supported on the Web at <http://www.software.ibm.com/data/db2/>. DB2 has an active and vibrant user community located on the Web at <http://www.idug.org>.

MySQL

The corporate resource for MySQL is <http://www.mysql.com>, and another good site is <http://theoryx5.uwinnipeg.ca/mysql>. A great developer resource with lots of useful tips is Devshed.com. See http://www.devshed.com/Server_Side/MySQL/ for MySQL-specific information.

PostgreSQL

The home for this open source database is located at <http://www.postgresql.org>. With a great deal of useful information available for download, this site also maintains mailing lists for PostgreSQL users. Another PostgreSQL site worth investigating is <http://www.pgsql.com>, which offers support for commercial customers.

Oracle

Oracle's cyberspace home is <http://www.oracle.com>. A great resource for hard-core Oracle users is <http://www.oracle.com/technology>. You can also find all Oracle documentation at <http://tahiti.oracle.com>.

SQL Server

The official Microsoft SQL Server web site is <http://www.microsoft.com/sql/>. Another good resource is found at the home of the Professional Association for SQL Server (PASS) at <http://www.sqlpass.org>.

Changes in the Second Edition

Although the ANSI standards committee released the current version of SQL in 2003, called SQL2003, this book has grown dynamically in a variety of new ways in response to request from our readers. In fact, when you compare the first and second editions of *SQL in a Nutshell*, you'll see that the second edition is, in reality, an entirely new book. Here are more details about changes in the second edition:

New format

We've developed an entirely new format that reduces redundancy and increases coverage for the SQL2003 standard and each vendor's implementation. Rather than use the expository style of the first edition, we've taken a keyword description style that breaks all the keywords and clauses into small, bite-sized entries.

New database platforms

We've added full coverage for IBM's DB2 UDB database platform running on Unix, Linux, and Windows. In addition, we've added an appendix that describes Sybase's implementation of the SQL commands, as they differ from Microsoft SQL Server's implementation.

Database programming

We've added a chapter that covers the basics of how programmers can interface their frontend programs to the backend database platforms covered in this book.

More complete coverage

We've added many more examples and added SQL commands not covered in the first edition. In addition, we've added much more complete coverage for SQL functions — in particular, vendor-specific functions that are not part of the SQL2003 standard.

More examples

You can never have too many examples. We've added examples for the most basic uses of the commands under the SQL2003 headings, with even more examples that highlight the unique and powerful extensions offered by each database platform.

Acknowledgments

We'd like to take a moment to thank a few special individuals at O'Reilly Media. First, we owe a huge debt of gratitude to Jonathan Gennick, the editor of the second edition. Jonathan kept us on track and on task, even when the cares of this world threatened to derail the project. Jonathan's attention to detail and exceptional management skills, in addition to his talents as an editor, are the reason this book is here today. Thank you! And of course, thanks to Tim O'Reilly for having a direct hand in the birth of this book.

We also owe a debt to our fine technical reviewers. To Peter Gulutzan (SQL Standard), Thomas Lockhart (PostgreSQL), Alan Beaulieu (Oracle), Baya Pavliachvili (Microsoft SQL Server), Bobby Fielding (DB2), Doug Doole (DB2), Rick Swagerman (DB2), Josh Stellan (DB2), Brian Lalonde (Database Programming), John Haydu (Oracle, Phani Arega (Oracle), and Paul DuBois (MySQL): we owe you a hearty thanks! Your contributions have greatly improved the accuracy, readability, and value of this book. Without you, our sections on each of the language extensions would have been on shaky ground. In addition, we'd like to tip our hat to Peter Gulutzan and Trudy Pelzer for their book *SQL-99 Complete, Really!*, which assisted our understanding of the ANSI SQL2003 standards.

Brand Hunt's Acknowledgments

To my wife Michelle, without your continued support and love, I wouldn't be a part of this project. I'm appreciative for every moment we've shared and your forgiveness for my keeping you awake at night with the "tappity-tap-taps" emanating from the computer.

To my parents Rex and Jackie, the two biggest influences in everything I've ever done correctly — especially those things that frequently take multiple attempts (like writing!)

A huge thanks to my teammates Kevin, Daniel, and Jonathan for letting me participate in this project and exercising so much patience tutoring a first time O'Reilly author. Your professionalism, work ethic, and ability to make the most tedious tasks fun is so admirable I plan to steal it and adopt it as my own!

Mad props to my friends and colleagues at Rogue Wave Software, ProWorks, NewCode Technology, and Systems Research and Development, for providing the ultimate sand box for refining skills in SQL, databases, business, software development, writing, and friendship: Gus Waters, Greg Koerper, Marc Manley, Wendi Minne, Erin Foley, Elaine Cull, Randall Robinson, Dave Ritter, Edin Zulic, David Noor, Jim Shur, Chris Mosbrucker, Dan Robin, Mike Faux, Jason Prothero, Tim Romanowski, Andy Mosbrucker, Jeff Jonas, Jeff Butcher, Charlie Barbour, Steve Dunham, Brian Macy, and Ze'ev Mehler.

Daniel Kline's Acknowledgments

I'd like to thank my brother, Kevin, for his continued willingness to work with me, to my colleagues at the University of Alaska Anchorage for their suggestions, and to the readers and users of the first edition of *SQL in a Nutshell* for their honest feedback and useful critiques. I'd also like to thank the Japanese translators of the first edition—Ryoko Akaike, O'Reilly Japan editor; Ishii Tatsuo, open source manager of Software Research Associates, Inc. and author of *PostgreSQL Perfect Guide*; and Toru Miyahara, CEO of BigiNet Co., Ltd.—for their careful review and helpful suggestions.

Kevin E. Kline's Acknowledgments

Many people helped deliver the big, thick book you hold in your hands. This note expresses our appreciation to those who helped make this a book reality.

First of all, Dan and Brand get a big slap on the back for working so hard to create the content you are now reading. Brand had to spend a lot of nights working on this both before and after his wedding. Thank you, Michelle, for lending us Brand when you wanted him the most.

Next, Jonathan Gennick, our editor at O'Reilly Media, gets a warm handshake and squeeze on the shoulder. You travailed through many extra months of hard labor bringing this book out of the shadows and into the light. I think there's a special blessing in store for those who've carried themselves with as much patience, professionalism, and good humor as you.

We must also express deep thanks and appreciation to our technical reviewers who caught many errors, omissions, and plain ol' mistakes and diverted us from many embarrassments:

- ANSI SQL — Peter Gulutzan, author of *SQL-99 Complete, Really* and *SQL Performance Tuning*
- DB2 — Doug Doole, Bobby Fielding, and Richard Swagerman of IBM; Josh Steffan of Quest Software

- MySQL — Paul DuBois, author of *MySQL*
- Oracle — Phani Arega of Quest Software and the reviewers at Oracle
- PostgreSQL — Thomas Lockhart, original author of the PostgreSQL open source documentation
- SQL Server — Baya Pavliashvili of Healthstream, Inc.
- Database Programming — Brian LaLonde

To all of my colleagues at Quest Software go a very big thanks for your support and encouragement. Rony Lerner, Deb Jenson, Eyal Aronoff, John Newsom, and Vinny Smith: thank you for investing your trust in me and making these last three years with Quest Software such a blast.

Thank you to all the guys on the SQL Server team for making our products the best on the market: John Theron, Patrick O'Keeffe, John Ortega, Mark Simon, Hasan Fahimi, Lee Grissom, Joe Motley, Adrian Tudor, Israel Kalush, Amit Kubovsky, and Ross Doering. I couldn't ask for a better team or a better set of friends.

Thanks to the SQL Server team at Microsoft for keeping me in the know: Euan Garden, Richard Waymire, Joe Yong, Don Peterson, Mark Sousa, Steven Dybing, Fernando Caro, Tom Rizzo, and Bill Baker. Your product rocks!

Finally, a word for my family. It's hard to work ridiculously long hours for months on end, between the day job and the book project, without making some sacrifices. And yet, somehow, we managed to spend so much good time together. Katie Jo, you've learned how to talk between the time I started and the time I finished. I still get choked up hearing you say "Yi yove you, daddy." Anna Lynn, your super powers rescued me plenty of times when I was down. I want to hear "Supergirl to da rescue!" the next time I need rescuing. Here's a secret: I keep that jar of fresh pistachios on my desk just because they're your favorite. Emily, thank you for letting me paint your nails. Some daughters don't let their dads that close. Dylan, you whipped me at so many games on the PS2 that I don't know why you still want me to play with you. But, one day, the ol' man is gonna take you down! Kelly, you are my tonic, my therapy, and my medicine. Just a touch or a glance or a smile from you and my night turns to day and my despair to hope. Thank you for loving me, supporting me, and seeing me through this project.

Chapter 1. SQL History and Implementations

In the early 1970s, the seminal work of IBM research fellow Dr. E. F. Codd led to the development of a relational data model product called SEQUEL, or Structured English Query Language. SEQUEL ultimately became SQL, or *Structured Query Language*.

IBM, along with other relational database vendors, wanted a standardized method for accessing and manipulating data in a relational database. Although IBM was first to develop relational database theory, Oracle was first to market with the technology. Over time, SQL proved popular enough in the marketplace to attract the attention of the American National Standards Institute (ANSI), which released standards for SQL in 1986, 1989, 1992, 1999, and 2003. Since 1986, competing languages have allowed programmers and developers to access and manipulate relational data. However, few were as easy to learn or as universally accepted as SQL. Programmers and administrators now have the benefit of learning a single language that, with minor adjustments, is applicable to a wide variety of database platforms, applications, and products.

SQL in a Nutshell, Second Edition, describes five common implementations of SQL2003:

- IBM's DB2 Universal Database Version 8 for Linux, Unix, and Windows
- MySQL Version 4
- Oracle Database 10g
- PostgreSQL Version 7
- Microsoft's SQL Server 2000

We also cover Sybase Adaptive Server Enterprise (ASE), but to a lesser degree, in [Appendix B](#).

1.1 The Relational Model and ANSI SQL

Relational Database Management Systems (RDBMSs), such as those covered in this book, are the primary engines of information systems worldwide, particularly web applications and distributed client/server computing systems. They enable a multitude of users to quickly and simultaneously access, create, edit, and manipulate data without impeding or impacting other users. They also allow developers to write useful applications to access their resources as well as provide administrators with the capabilities they need to maintain, secure, and optimize organizational data resources.

An RDBMS is defined as a system whose users view data as a collection of tables related to each other through common data values. Data is stored in *tables*, which are composed of *rows* and *columns*. Tables of independent data can be linked (or *related*) to one another if they each have unique, identifying columns of data (called *keys*) that represent a data value held in common. E. F. Codd first described relational database theory in his landmark paper "A Relational Model of Data for Large Shared Data Banks," published in the *Communications of the ACM* (Association for Computing Machinery) in June, 1970. Under Codd's new relational data model, data was structured (into tables of rows and columns); manageable using operations like selections, projections, and joins; and consistent as the result of integrity rules like keys and referential integrity. Codd also articulated rules that governed how a relational database should be designed; the process for applying these rules is now known as *normalization*.

1.1.1 Codd's Rules for Relational Database Systems

Codd applied rigorous mathematical theories, primarily set theory, to the management of data, and he compiled a list of criteria a database must meet to be considered relational. At its core, the relational database concept centers around storing data in tables. This concept is now so common as to seem trivial; however, not long ago designing a system capable of sustaining the relational model was considered a long shot with limited usefulness. Following are Codd's *Twelve Principles of Relational Databases*:

1. Information is represented logically in tables.
2. Data must be logically accessible by table, primary key, and column.
3. Null values must be uniformly treated as "missing information," not as empty strings, blanks, or zeros.
4. Metadata (data about the database) must be stored in the database just as regular data is.
5. A single language must be able to define data, views, integrity constraints, authorization, transactions, and data manipulation.
6. Views must show the updates of their base tables and vice versa.
7. A single operation must be available to do each of the following operations: retrieve data, insert data, update data, or delete data.
8. Batch and end-user operations are logically separate from physical storage and access methods.
9. Batch and end-user operations can change the database schema without having to recreate it or the applications built upon it.
10. Integrity constraints must be available and stored in the metadata, not in an application program.
11. The data manipulation language of the relational system should not care where or how the physical data is distributed and should not require alteration if the physical data is centralized or distributed.
12. Any row processing done in the system must obey the same integrity rules and constraints that set-processing operations do.

These principles continue to be the litmus test used to validate the "relational" characteristics of a database platform; a database that does not meet all of these rules is not fully relational. While these rules do not apply to applications development, they do determine whether the database engine itself can be considered truly "relational." Currently, most commercial RDBMS products pass Codd's test. Among the platforms discussed in *SQL in a Nutshell, Second Edition*, only MySQL fails to support all of these requirements.

Understanding Codd's principles assists programmers and developers in the proper development and design of Relational Databases (RDBs). The following sections detail how some of these requirements are met within SQL using RDBs.

1.1.1.1 Data structures (rules 1, 2, and 8)

Codd's rules 1 and 2 state that "information is represented logically in tables" and that "data must be logically

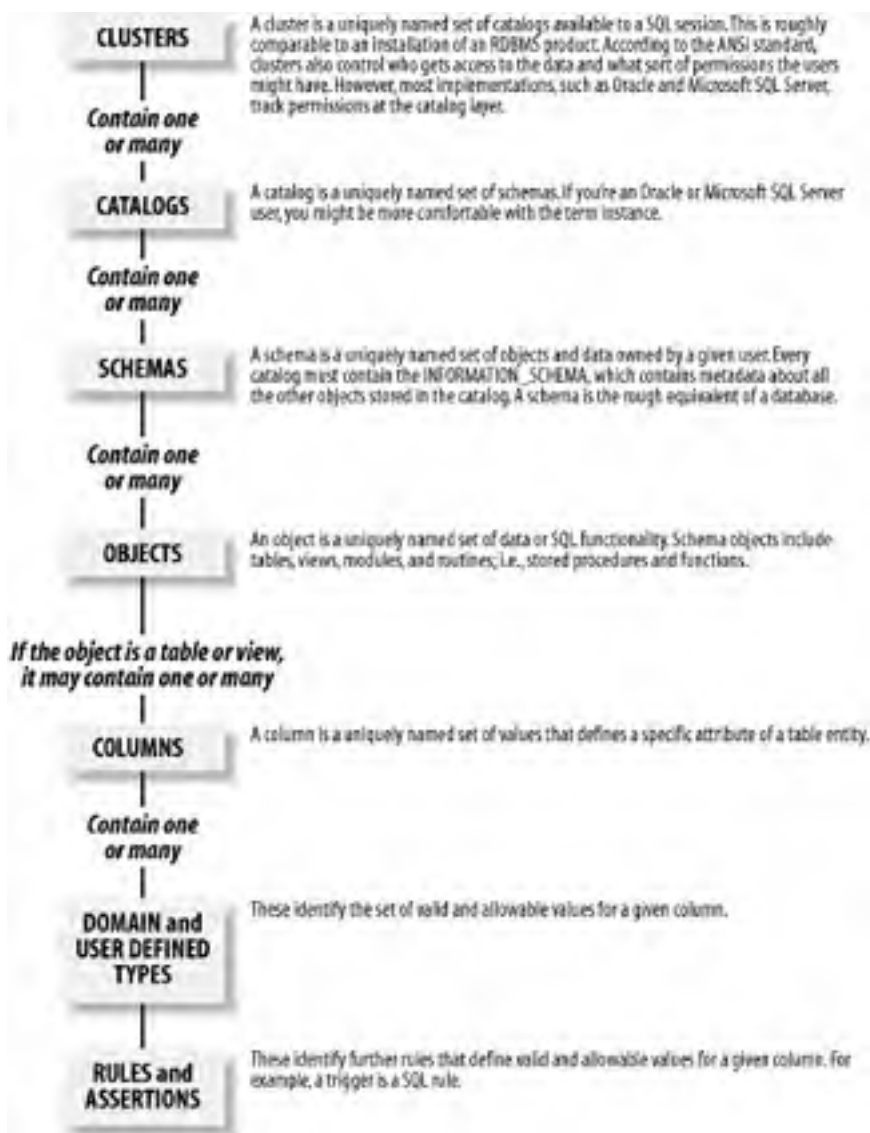
accessible by table, primary key, and column." So the process of defining a table for a SQL database does not require that programs instruct the database how to interact with the underlying physical data structures. Furthermore, SQL logically isolates the process of accessing data and physically maintaining that data as required by rule 8, "batch and end-user operations are logically separate from physical storage and access methods."

In the relational model, data is shown logically as a two-dimensional *table* that describes a single entity (for example, business expenses). Academics refer to tables as *entities* and to columns as *attributes*. Tables are composed of *rows*, or *records* (academics call them *tuples*), and *columns* (called *attributes*, since each column of a table describes a specific attribute of the entity). The intersection of a record and a column provides a single *value*. The column or columns whose values uniquely identify each record can act as a *primary key*. These days this representation seems elementary, but it was actually quite innovative when it was first proposed.

SQL2003 defines a whole data structure hierarchy beyond simple tables, though tables are the core data structure. Relational design handles data on a table-by-table basis, not on a record-by-record basis. This table-centric orientation is the heart of set programming. Consequently, almost all SQL commands operate much more efficiently against sets of data within or across tables than against individual records. Said another way, effective SQL programming requires that you think in terms of sets of data, rather than of individual rows.

Figure 1-1 is a description of the SQL2003 terminology used to describe the hierarchical data structures used by a relational database: *clusters* contain sets of *catalogs*; *catalogs* contain sets of *schemas*; *schemas* contain sets of *objects*, such as *tables* and *views*; and *tables* are composed of sets of *columns* and *records*.

Figure 1-1. SQL2003 dataset hierarchy




For example, in a **Business_Expense** table, a column called **Expense_Date** might show when an expense was

incurred. Each record in the table describes a specific entity; in this case, everything that makes up a business expense (when it happened, how much it cost, who incurred the expense, what it was for, and so on). Each attribute of an expense, in other words each column, is supposed to be *atomic*; that is, each column is supposed to contain one, and only one, value. If a table is constructed in which the intersection of a row and column can contain more than one distinct value, then one of SQL's primary design guidelines has been violated. (Naturally, some of the database platforms discussed in this book allow you to place more than one value into a column, via *VARRAY* or *TABLE* datatypes.)

There are rules of behavior specified for column values. Foremost is that column values must share a common *domain*, better known as a *datatype*. For example, the value *ELMER* should not be placed into the **Expense_Date** field. The value *ELMER* is a string, not a date, while the **Expense_Date** field can contain only dates. Therefore, this column would be defined as having a *DATE* datatype. In addition, SQL2003 allows further controls of such values through the application of *constraints* and *assertions*. (Constraints are discussed in detail later in [Chapter 2](#).) A SQL constraint might limit **Expense_Date** to expenses less than a year old.

Additionally, data access for all individuals and computer processes is controlled at the schema level by an *AuthorizationID* or *user*. Permissions to specific sets of data may be granted or restricted to each user.


Moreover, SQL databases also employ *character sets* and *collations*. Character sets are the "symbols" or "alphabets" used by the "language" of the data. For example, the American English character set does not contain the special character for ñ in the Spanish character set. Collations are sets of sorting rules that operate on a character set. A collation defines how a given data manipulation operation sorts data. For example, an American English character set might be sorted either by *character-order*, *case-insensitive*, or by *character-order*, *case-sensitive*.

 The ANSI standard does not say how sorts should be done, only that platforms must provide common collations found in a particular language.

It is important to know what collation you are using when writing SQL code against a database platform since it can have a direct impact on how queries behave, particularly on the behavior of the *WHERE* and *ORDER BY* clauses of *SELECT* statements. For example, a query that sorts data using a binary collation will return data in a very different order than one that sorts data using, say, an American English collation.

1.1.1.2 NULLS (rule 3)

Most databases allow any of their supported datatypes to store NULL values. Inexperienced SQL programmers and developers tend to think of NULL as zero or blank. In fact, NULL is neither of these. In SQL2003, NULL literally means that the value is unknown or indeterminate. (This question alone—whether NULL should be considered unknown or indeterminate—is the subject of academic debate.) This differentiation enables a database designer to distinguish between those entries that represent a deliberately placed zero (for example) and those where either the data is not recorded in the system or where a NULL has been explicitly entered. For an example of this semantic difference, consider a system that tracks payments. A product with a NULL price does not mean that the product is free; instead, a NULL price indicates that the amount is not known or perhaps not yet determined.

 There is a good deal of differentiation between the database platforms in terms of how they handle NULL values. This leads to some major porting issues between those platforms relating to NULLs. For example, an empty string (i.e., a NULL string) is inserted as a NULL value on Oracle. With the exception of Sybase, all the other databases covered in this book permit the insertion of an empty string into *VARCHAR* and *CHAR* columns.

One side effect of the indeterminate nature of a NULL value is that it cannot be used in a calculation or a comparison. Here are a few brief but very important rules, from the ANSI standard, to remember about the behavior of NULL values when dealing with NULLs in SQL statements:

- A NULL value cannot be inserted into a column defined as NOT NULL.
- NULL values are not equal to each other. It is a frequent mistake to compare two columns that contain NULL and expect the NULL values to match. (The proper way to identify a NULL value in a *WHERE* clause or in a Boolean expression is to use phrases such as "value IS NULL" and "value IS NOT NULL".)
- A column containing a NULL value is ignored in the calculation of aggregate values such as *AVG*, *SUM*, or *MAX.COUNT*.
- When columns that contain NULL values are listed in the *GROUP BY* clause of a query, the query output

contains a single row for NULL values. In essence, the ANSI standard considers all NULLs found to be in a single group.

- *DISTINCT* and *ORDER BY* clauses, like *GROUP BY*, also see NULL values as indistinguishable from each other. With the *ORDER BY* clause, the vendor is free to choose whether NULL values sort high (first in the result set) or sort low (last in the result set) by default.

1.1.1.3 Metadata (rules 4 and 10)

Codd's fourth rule for relational databases states that data about the database (metadata) must be stored in standard tables, just as all other data. *Metadata* is data that describes the database itself. For example, every time you create a new table or view in a database, records are created and stored that describe the new table. Additional records are needed to store any columns, keys, or constraints on the table. This technique is implemented in most commercial and open source SQL database products. For example, SQL Server uses what it calls "system tables" to track all the information about the databases, tables, and database objects in any given database. It also has "system databases" that keep track of information about the server on which the database is installed and configured.

1.1.1.4 The language (rules 5 and 11)

Codd's rules do *not* require SQL to be used with a relational database. His rules, particularly rules 5 and 11, only specify how the language should behave when coupled with a relational database. At one time, SQL competed with other languages (like Digital's RDO or Fox/PRO) that might have fit the relational bill, but SQL won out for three reasons. First, SQL is a relatively simple, intuitive, English-like language that handles most aspects of data manipulation. Second, SQL is satisfyingly high-level. A programmer or Database Administrator (DBA) does not have to spend time ensuring that data is stored in the proper memory registers or that data is cached to disk. The Database Management System (DBMS) handles that task automatically. Finally, because SQL is not owned by any single vendor, it was adopted across a number of platforms.

1.1.1.5 Views (rule 6)

A *view* is a virtual table that does not exist as a physical data repository of data, but is instead constructed, on the fly, from a *SELECT* statement whenever the view is queried. Views enable you to construct different representations out of the same source data for a variety of audiences without having to alter the way in which the data is stored.



Some vendors support database objects called *materialized views*. Materialized views are not governed by the same rules as ANSI standard views.

1.1.1.6 Set operations (rules 7 and 12)

Other database manipulation languages, such as the venerable Xbase, perform their data operations quite differently from SQL. These languages require you to tell the program exactly how to treat the data, one record at a time. Since the program cycles down through a list of records, performing its logic on one record after another, this style of programming is frequently called *row processing* or *procedural programming*.

In contrast, SQL programs operate on logical *sets* of data. Set theory is applied in most all SQL statements such as *SELECT*, *INSERT*, *UPDATE*, or *DELETE* statements. In effect, data is selected from a set called a table. Unlike the row processing style, *set processing* allows a programmer to tell the database simply *what* is required, not *how* each individual piece of data should be handled. Sometimes set processing is referred to as *declarative processing*, since a programmer declares only what data is wanted, as in "Give me all employees in the southern region who earn more than \$70,000 per year," rather than describing the exact procedure used to retrieve or manipulate the data.



Set theory was the brainchild of mathematician Georg Cantor, who developed it at the end of the nineteenth century. At the time, set theory (and his theory of the infinite) was quite controversial. Today, set theory is such a common part of life that it is learned in elementary school.

Examples of set theory in conjunction with relational databases are detailed in the following section.

1.1.2 Codd's Rules in Action: Simple SELECT Examples

Up to this point, the chapter has been about the individual aspects of a relational database platform as defined by Codd and implemented under ANSI SQL. This section presents a high-level overview of the most important SQL statement, *SELECT*, and some of its most salient points—namely, the relational operations known as *projections*, *selections*, and *joins*:

Projection

Retrieves specific columns of data.

Selection

Retrieves specific rows of data.

Join

Returns columns and rows from two or more tables in a single result set.

Although at first glance it might appear that the *SELECT* statement deals only with the relational selection operation, in actuality, *SELECT* embodies all three operations. (Refer to [SELECT Statement](#).)

The following statement embodies the projection operation by selecting the first and last names of an author, plus his home state, from the **authors** table:

```
SELECT au_fname, au_lname, state
FROM authors
```

The results from any such *SELECT* statement are presented as another table of data:

au_fname	au_lname	state
Johnson	White	CA
Marjorie	Green	CA
Cheryl	Carson	CA
Michael	O'Leary	CA
Meander	Smith	KS
Morningstar	Greene	TN
Reginald	Blotchet-Halls	OR
Innes	del Castillo	MI

The resulting data is sometimes called a *result set*, *work table*, or a *derived table*, differentiating it from the *base table* in the database that is the target of the *SELECT* statement.

It is important to note that the relational operation of projection, and not selection, is specified using the *SELECT* clause (that is, the keyword *SELECT* followed by a list of expressions to be retrieved) of a *SELECT* statement. Selection, the operation of retrieving specific rows of data, is specified using the *WHERE* clause in a *SELECT* statement. *WHERE* filters out unwanted rows of data and retrieves only the requested rows. Using the previous example, the following selects authors from states other than California:

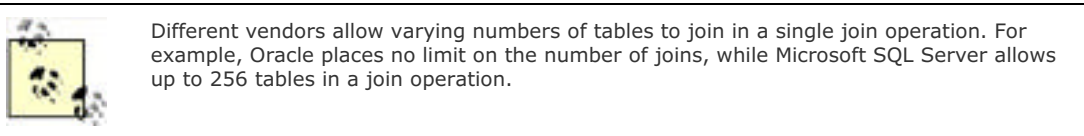
```
SELECT au_fname, au_lname, state
FROM authors
WHERE state <> 'CA'
```

The previous query retrieved all authors; the result of this query is a much smaller subset of records:

au_fname	au_lname	state
Meander	Smith	KS
Morningstar	Greene	TN
Reginald	Blotchet-Halls	OR
Innes	del Castillo	MI

By combining the capabilities of projection and selection together in a single query, you can use SQL to retrieve only the columns and records that you need at any given time.

Joins are the next, and last, relational operation we're going to talk about in this section. A join relates one table to another in order to return a result set consisting of related data from both tables.



The ANSI standard method of performing joins is to use the *JOIN* clause in a *SELECT* statement. An older method, known as a *theta join*, performs the join analysis in the *WHERE* clause. The following example shows both approaches. Each statement retrieves employee information from the **employee** base table as well as job descriptions from the **jobs** base table. The first *SELECT* uses the newer, ANSI *JOIN* clause, while the second *SELECT* uses a theta join:

-- ANSI style

```
SELECT a.au_fname, a.au_lname, t.title_id
FROM authors AS a
JOIN titleauthor AS t ON a.au_id = t.au_id
WHERE a.state <> 'CA'
```

-- Theta style

```
SELECT a.au_fname, a.au_lname, t.title_id
FROM authors AS a,
      titleauthor AS t
WHERE a.au_id = t.au_id
      AND a.state <> 'CA'
```

For more information about joins, refer to [JOIN Subclause](#).

1.2 History of the SQL Standard

Because SQL dialects began to proliferate, the ANSI published its first SQL standard in 1986 to bring greater conformity among vendors, and a second, widely-adopted standard in 1989. The International Standards Organization (ISO) also approves the SQL standard. ANSI released updates in 1992, known as SQL92 and SQL2, and again in 1999, termed both SQL99 and SQL3. Now that the ANSI standard has advanced, we generally use the industry standard term SQL2003. Each time it revises the SQL standard, ANSI adds new features and incorporates new commands and capabilities into the language. For example, the SQL99 standard added a group of capabilities that handled object-oriented datatype extensions.

1.2.1 What's New in SQL2003

SQL99 had two main parts, Foundation:1999 and Bindings:1999. The SQL2003 Foundation section includes all of the Foundation and Bindings standards from SQL99, but a new section called Schemata was created.

The Core requirements of SQL2003 did not change from Core SQL99. So the database platforms that conformed to Core SQL99 automatically conform to SQL2003. Although the Core of SQL2003 has no additions (except for a few new reserved words), a number of individual statements and behaviors have been updated or modified. Because these updates are reflected in the individual syntax descriptions of each statement in [Chapter 3](#), we won't spend time on them here.

A few elements of the Core in SQL99 have been deleted from SQL2003, including:

- The *BIT* and *BIT VARYING* datatypes
- The *UNION JOIN* clause
- The *UPDATE...SET ROW* statement

In addition, a number of features, most of which were rather obscure, have been added, deleted, or renamed. At this point in time, many of the new features of SQL2003 standard are interesting mostly from an academic standpoint because none of the database platforms support the feature yet. However, there are a few new features that hold more than passing interest:

Elementary OLAP functions

Adds an Online Analytical Processing (OLAP) amendment, including a number of windowing functions to support widely used calculations such as moving averages and cumulative sums. Windowing functions are aggregates computed over a window of data: *ROW_NUMBER*, *RANK*, *DENSE_RANK*, *PERCENT_RANK*, and *CUME_DIST*. OLAP functions are fully described in T611 of the standard. Some database platforms are starting to support the OLAP functions. Refer to [Chapter 4](#) for details.

Sampling

Adds the *TABLESAMPLE* clause to the *FROM* clause. This is useful for statistical queries on large databases, such as a data warehouse.

Enhanced numeric functions

Adds a large number of numeric functions. In this case, the standard was mostly catching up with the trend in the industry, since the new functions are already supported by one or more of the database platforms. Refer to [Chapter 4](#) for details.

Another important change from SQL92 is that SQL99 is built upon SQL92's *levels of conformance*.

1.2.2 Levels of Conformance

SQL92 first introduced levels of conformance by defining three categories: *Entry*, *Intermediate*, and *Full*. Vendors had to achieve at least Entry-level conformance to claim ANSI SQL compliance. The U.S. National Institute of Standards and Technology (NIST) later added the *Transitional* level between the Entry and Intermediate levels. So NIST's levels of conformance were Entry, Transitional, Intermediate, and Full, while ANSI's were only Entry, Intermediate, and Full. Each higher level of the standard was a superset of the subordinate level, meaning that each higher level of the standard included all the features of the lower level of conformance.

Later, SQL99 altered the base levels of conformance. Gone were the Entry, Intermediate, and Full levels of conformance. With SQL99, vendors must implement all the features of the lowest level of conformance, Core SQL99, in order to claim (and publish) that they are SQL99 ready. Core SQL99 includes the old Entry SQL92 feature set, features from other SQL92 levels, and some brand new features. A vendor may also choose to implement additional feature packages described in the SQL99 standard.

1.2.3 Supplemental Features Packages in the SQL2003 Standard

The SQL2003 standard represents the ideal, but very few vendors currently meet or exceed the Core SQL2003 requirements. The Core standard is like the interstate speed limit: some drivers go above, others go below, but few go exactly the speed limit. Similarly, vendor implementations can vary greatly.

The nine supplemental features packages, representing different subsets of commands, are platform-optional. Some features might show up in multiple packages, while others do not appear in any of the packages. These packages and their features are described in the list that follows.

Two committees—one within ANSI, the other within ISO, and both composed of representatives from virtually every RDBMS vendor—drafted the definitions shown in the list. In this collaborative and somewhat political environment, vendors compromised on exactly which proposed features and implementations were incorporated into a new standard.

Many times, a new feature in the ANSI standard is derived from an existing product or is the outgrowth of new research and development in the academic community. Consequently, vendor adoption of specific ANSI standards can be spotty at times. A relatively new addition to the SQL2003 standard is SQL/XML. The other parts of the SQL99 standard are persisted in SQL2003, though their names may have changed or they may have been slightly rearranged.

Part 1 - SQL/Framework

Includes common definitions and concepts used throughout the standard. Defines the way the standard is structured and how the various parts relate to one another. Describes the conformance requirements set out by the standards committee.

Part 2 - SQL/Foundation

Includes the Core, an augmentation of the SQL99 Core, and is the largest and most important part of the standard.

Part 3 - SQL/CLI (Call-Level Interface)

Defines the call-level interface for dynamically invoking SQL statements from external application programs. SQL/CLI also includes over 60 routine specifications to facilitate the development of truly portable shrink-wrapped software.

Part 4 - SQL/PSM (Persistent Stored Modules)

Standardizes procedural language constructs similar to those found in database platform-specific SQL dialects like PL/SQL and Transact-SQL.

Part 9 - SQL/MED (Management of External Data)

Defines the management of data located outside of the database platform using datalinks and a wrapper interface.

Part 10 - SQL/OBJ (Object Language Binding)

Describes how to embed SQL statements in Java programs. It is closely related to JDBC, but offers a few advantages over JDBC. It is also very different from the traditional host language binding possible in early versions of the standard.

Part 11 - SQL/Schemata

Defines over 85 views (three more than in SQL99) used to describe the metadata of each database and stored in a special schema called *INFORMATION_SCHEMA*. A number of views that existed in SQL99 have been updated.

Part 12 - SQL/JRT (Java Routines and Types)

Defines a number of SQL routines and types using the Java programming language. Features of Java, such as Java static methods and Java classes, are now supported.

Part 14 - SQL/XML

Adds a new type, called XML, four new operators (*XMLPARSE*, *XMLSERIALIZE*, *XMLROOT*, and *XMLCONCAT*), several new functions (described in [Chapter 4](#)), and the new *IS DOCUMENT* predicate. It also includes rules for mapping SQL-related elements (like identifiers, schemas, and objects) to XML-related elements.

Note that parts 5, 6, 7, and 8 do not exist by design.

Be aware that an RDBMS platform may claim SQL2003 compliance by meeting Core SQL99 standards, so read the vendor's fine print for a full description of its ANSI conformity features. By understanding what features comprise the nine packages, you gain a clear idea of both the capabilities of a particular RDBMS and how the various features behave when SQL code is transported to other database products.

The ANSI standards—which cover retrieval, manipulation, and management of data in commands, such as *SELECT*, *JOIN*, *ALTER TABLE*, and *DROP*—formalize many SQL behaviors and syntax structures across a variety of platforms. These standards become even more important as open source database products, such as MySQL and PostgreSQL, grow in popularity and are developed by virtual teams rather than large corporations.

SQL in a Nutshell, Second Edition, explains the SQL implementation of five popular RDBMSs. These vendors do not meet all the SQL2003 standards; in fact, all RDBMS platforms play a constant game of tag with the standards bodies. Many times, as soon as vendors close in on the standard, the standards bodies update, refine, or otherwise change the benchmark. Conversely, the vendors often implement new features that are not yet a part of the standard.

1.2.4 SQL2003 Statement Classes

Comparing statement classes further delineates SQL2003 from SQL92. However, you will still hear these terms bantered about. So you need to know them. In SQL92, SQL statements are grouped into three broad categories:

Data Manipulation Language (DML)

Provides specific data-manipulation commands such as *SELECT*, *INSERT*, *UPDATE*, and *DELETE*.

Data Definition Language (DDL)

Contains commands that handle the accessibility and manipulation of database objects, including *CREATE* and *DROP*.

Data Control Language (DCL)

Contains the permission-related commands *GRANT* and *REVOKE*.

In contrast, SQL2003 supplies seven core categories, now called *classes*, that provide a general framework for the types of commands available in SQL. These statement "classes" are slightly different than the SQL92 statement categories, since they attempt to identify the statements within each class more accurately and logically. Furthermore, because SQL is constantly under development, new features and commands enter the standard and may necessitate new statement classes. So, SQL2003 has a new sets of statement classes, originally added in the SQL99 standard, making them somewhat more comprehensible and logical. Additionally, the new statement classes now allow some "orphaned" statements—which did not fit well into any of the old categories—to be properly classified.

[Table 1-1](#) identifies the SQL2003 statement classes and lists some commands in each class, each of which is fully discussed later. At this point, the key is to remember the statement class title.

Table 1-1. SQL2003 statement classes

Class	Description	Example commands
SQL connection statements	Start and end a client connection	<i>CONNECT</i> , <i>DISCONNECT</i>
SQL control statements	Control the execution of a set of SQL statements	<i>CALL</i> , <i>RETURN</i>
SQL data statements	May have a persistent and enduring effect upon data	<i>SELECT</i> , <i>INSERT</i> , <i>UPDATE</i> , <i>DELETE</i>

SQL diagnostic statements	Provide diagnostic information and raise exceptions and errors	<i>GET DIAGNOSTICS</i>
SQL schema statements	May have a persistent and enduring effect on a database schema and objects within that schema	<i>ALTER, CREATE, DROP</i>
SQL session statements	Control default behavior and other parameters for a session	<i>SET statements like SET CONSTRAINT</i>
SQL transaction statements	Set the starting and ending point of a transaction	<i>COMMIT, ROLLBACK</i>

Those who work with SQL regularly should become familiar with both the old (SQL92) and the new (SQL2003) statement classes, since both nomenclatures are still used to refer to SQL features and statements.

1.3 SQL Dialects

The constantly evolving nature of the SQL standard has given rise to a number of SQL *dialects* among the various vendors and platforms. These dialects most commonly evolved because the user community of a given database vendor required capabilities in the database before the ANSI committee created a standard. Occasionally, though, a new feature is introduced by the academic or research communities due to competitive pressures from competing technologies. For example, many database vendors are augmenting their current programmatic offerings with Java (as is the case with DB2, Oracle, and Sybase) or VBScript (as Microsoft is doing). In the future, programmers and developers will use these programming languages in concert with SQL to build SQL programs.

Many of these dialects include conditional processing capabilities (such as that controlled through *IF . . . THEN* statements), control-of-flow functions (such as *WHILE* loops), variables, and error handling. Because ANSI had not yet developed a standard for these important features at the time users began to demand them, RDBMS developers and vendors were free to create their own commands and syntax. In fact, some of the earliest vendors from the 1980s have variances in the most elementary commands, such as *SELECT*, because their implementations predate the standards. ANSI is now refining standards that address these inconsistencies.

Some of these dialects have introduced procedural commands to support the functionality of a much more complete programming language. For example, these procedural implementations contain error-handling commands, control-of-flow language, conditional commands, variable handling, arrays, and many other extensions. Although these are technically divergent procedural implementations, they are called dialects here. The SQL/PSM (Persistent Stored Module) package provides many features associated with programming stored procedures and incorporates many of the extensions offered by these dialects.

Some popular dialects of SQL include:

PL/SQL

Found in Oracle. PL/SQL stands for Procedural Language/SQL and contains many similarities to the language Ada.

Transact-SQL

Used by both Microsoft SQL Server and Sybase Adaptive Server. As Microsoft and Sybase have moved away from the common platform they shared early in the 1990s, their implementations of Transact-SQL have also diverged.

PL/pgSQL

The name of the SQL dialect and extensions implemented in PostgreSQL. The acronym stands for Procedural Language/postgreSQL.

SQLPL

The newest dialect is DB2's SQLPL (SQLProcedural Language), which is based on the standard SQL control statements. Most of the other dialects predate the standard, meaning you'll find a lot of variations from the SQL standard. But since SQLPL came after the standard, it is in greater compliance.

If you plan to work extensively with a single database system, you should learn the intricacies of your preferred SQL dialect or platform.

Chapter 2. Foundational Concepts

SQL provides an easy, intuitive way to interact with a database. While the SQL2003 standard does not define the concept of a "database," it does define all the functions and concepts needed for a user to create, retrieve, update, and delete data. It is important to know the types of syntax in the SQL2003 standard and the particular platform-specific syntax guidelines.

2.1 Database Platforms Described in This Book

SQL in a Nutshell, Second Edition, describes the SQL standard and the platform-specific implementations of five leading RDBMSs:

DB2

IBM's popular DBMS runs on hardware platforms ranging from PCs all the way up to huge mainframes. It runs on a multitude of operating systems, including the burgeoning Linux OS. DB2 is very popular in corporate environments, especially those with a healthy investment in IBM software, equipment, or services. This book covers DB2 Universal Database Version 8.1 for Linux, Unix, and Windows.

MySQL

MySQL is a popular open source DBMS that is known for its blistering performance. It runs on numerous operating systems, including most Linux variants. To improve performance, it has a slimmer feature set than do many other DBMSs. This book covers MySQL 4.0.

Oracle

Oracle is a leading RDBMS in the commercial sector. It runs on a multitude of operating systems and hardware platforms. Its scalable, reliable architecture has made it the platform of choice for many users. *SQL in a Nutshell*, Second Edition, covers Oracle Database 10g.

PostgreSQL

PostgreSQL is the most feature-rich open source database platform available. Where MySQL is best known for its high-speed performance, PostgreSQL is best known for its excellent support for ANSI standards and robust transaction processing capabilities, as well as rich datatype and database object support. In addition to its full set of features, PostgreSQL runs on a wide variety of operating systems and hardware platforms. This book covers PostgreSQL 7.2.

SQL Server

Microsoft SQL Server is a popular RDBMS that runs only on the Windows platform. Its features include ease of use, an all-inclusive feature set, low cost, and high performance. This book covers Microsoft SQL Server 2000.

In addition to the five databases just mentioned, Sybase Adaptive Server Enterprise (ASE) Version 12.5 is covered separately in [Appendix A](#). Because Sybase and SQL Server have common roots, the appendix discusses the differences between the generally usable SQL Server syntax and those specifics and extensions offered by Sybase ASE.

2.2 Categories of Syntax

To begin to use SQL, you must understand how statements are written. SQL syntax falls into four main categories. Each category is introduced in the following list and then explained in further detail in the sections that follow:

Identifiers

Describe a user- or system-supplied name for database objects, such as a database, a table, a constraint on a table, the columns in a table, a view, etc.

Literals

Describe a user- or system-supplied string or value that is not otherwise an identifier or a keyword. Literals may be strings like "hello", numbers like 1234, dates like "Jan 01, 2002", or Boolean values like TRUE.

Operators

Are symbols specifying an action performed on one or more expressions, most often in *DELETE*, *INSERT*, *SELECT*, or *UPDATE* statements. Operators are also used frequently in the creation of database objects.

Reserved words and keywords

Have special meaning to the database SQL parser, such as *SELECT*, *GRANT*, *DELETE*, or *CREATE*. *Reserved words* are words that cannot be used as identifiers within the database platform, usually commands or SQL statements. *Keywords* are words that may become reserved some time in the future. You can circumvent the restriction on using reserved words and keywords as identifiers through *quoted identifiers*, described in a moment. This is not recommended! Elsewhere in the book (besides here), we use the term *keyword* to describe both concepts.

2.2.1 Identifiers

Keep in mind that RDBMSs are built upon set theory. In ANSI terms, clusters contain sets of catalogs, catalogs contain sets of schemas, schemas contain sets of objects, and so on. Most database platforms use corollary terms: *instances* contain one or more databases; *databases* contain one or more schemas; *schemas* contain one or more tables, views, or stored procedures, and the privileges associated with each object. At each level of this structure, items require a unique name (that is, an identifier) so that they can be referenced by programs and system processes. This means that each *object* (whether a database, table, view, column, index, key, trigger, stored procedure, or constraint) in a RDBMS must be identified. When issuing the command that creates a database object, you must specify an identifier (i.e., a name) for that new object.

There are two important categories of rules that experienced programmers keep in mind when choosing an identifier for a given item:

Naming conventions

Include logical rules of thumb or naming conventions that ultimately create better database structures and data tracking. These are not so much required by SQL as they are the distilled experience of practiced programmers.

Identifier rules

Are those set by the SQL standard and implemented by the platforms. Such rules include characteristics like how long a name may be. These identifier conventions are covered for each vendor later in this chapter.

2.2.1.1 Naming conventions

Naming conventions establish a standard baseline for choosing object identifiers. In this section, we show a list of naming conventions (rules for picking your identifiers) that are based on long years of experience. The SQL standard has no comment on naming conventions outside of the uniqueness of an identifier, its length, and the characters that are valid within the identifier:

Select a name that is meaningful, relevant, and descriptive

Do not name a table **XP03**; instead, name it **Expenses_2005**, showing that it stores expenses for the year 2005. Remember that other people will likely be using the table or database too, perhaps long after you have gone, and the names should make sense at a glance. Each database vendor has limits on object name size, but names generally may be long enough to make sense to anyone reading them.

Choose and apply the same case throughout

Use either all uppercase or all lowercase for all objects throughout the database. Some database servers are case-sensitive, so using mixed-case identifiers might cause problems later.

Use abbreviations consistently

Once an abbreviation has been chosen, it should be used consistently throughout the database. For example, if EMP is used as an abbreviation for EMPLOYEE, then EMP should be used throughout the database. Do not use EMP in some places and EMPLOYEE in others.

Use complete, descriptive, meaningful names with underscores for reading clarity

A column name **UPPERCASEWITHUNDERScores** is not as easy to read as **UPPERCASE_WITH_UNDERScores**.

Do not put company or product names in database object names

Companies get acquired and products change names. These elements are too transitory to be included in database object names.

Do not use overly obvious prefixes or suffixes

For example, don't use "DB_" as a prefix for a database, and don't prefix every view with "V_". Simple queries to the system table of the database can tell the DBA or database programmer what type of object an identifier represents.

Do not fill up all available space for the object name

If the database platform allows a 32-character table name, try to leave at least a few free characters at the end. Some database platforms sometimes append prefixes or suffixes to table names when manipulating temporary copies of the tables.

Do not use quoted identifiers

Quoted identifiers are object names stored within double-quotes. (The ANSI standard calls these *delimited identifiers*.) Encapsulating an identifier within double-quotes allows you to create names that will be difficult to use and that may cause you problems later. Quoted identifiers are also case-sensitive. For example, you could embed spaces, special characters, mixed case characters, or even escape sequences within a quoted identifier, but because some third-party tools (and even vendor-supplied tools) cannot handle special characters in names, you should not make extensive use of quoted identifiers. Some platforms allow other delimiting symbols than double-quotes. For example, SQL Server uses brackets [] to designate quoted identifiers.

There are several benefits to following these naming conventions. First, your SQL code becomes, in a sense, self-documenting because the chosen names are meaningful and understandable to other users. Second, your SQL code and database objects are easier to maintain—especially for other users who come after you—because your objects are consistently named. Finally, maintaining consistency increases database functionality. If the database ever has to be transferred or migrated to another RDMS, consistent and descriptive naming saves both time and energy. Giving a few minutes of thought to naming SQL objects in the beginning can prevent problems later on.

2.2.1.2 Identifier rules

Identifier rules are rules for identifying objects, enforced by the database platforms, within the database. These rules apply to normal identifiers, not quoted identifiers. Rules specified by the SQL2003 standard generally differ somewhat from those of a specific database vendor. [Table 2-1](#) contrasts the SQL2003 rules with those of the five RDMS platforms covered in this book.

Table 2-1. Platform-specific rules for regular object identifiers (excludes quoted identifiers)

Characteristic	Platform	Specification
Identifier size	SQL2003	128 characters
	DB2	128 characters, depending on the object
	MySQL	64 characters
	Oracle	30 bytes (number of characters depends on the character set); database names are limited to 8 bytes
	PostgreSQL	31 characters (NAMEDATALEN property minus 1)
	SQL Server	128 characters (temp tables are limited to 116 characters)
Identifier may contain	SQL2003	Any number, character, or underscore
	DB2	Any number, uppercase character, digit, or the underscore character
	MySQL	Any number, character, or symbol
	Oracle	Any number, character, and the underscore (_), pound (#), and dollar (\$) symbols
	PostgreSQL	Any number, character, or the underscore (_) symbol
	SQL Server	Any number, character, and the underscore (_), at sign (@), pound (#), and dollar (\$) symbols
Identifier must begin with	SQL2003	A letter
	DB2	A letter
	MySQL	A letter or number (but cannot be composed entirely of numbers)
	Oracle	A letter
	PostgreSQL	A letter or underscore (_)
	SQL Server	A letter, underscore (_), at sign (@), or pound (#)
Identifier cannot contain	SQL2003	Spaces or special characters
	DB2	Spaces or special characters
	MySQL	Period (.), slash (/), or ASCII(0) and ASCII(255). Quote (') and double-quote (") are only allowed in quoted identifiers.
	Oracle	Spaces, double-quotes ("), or special characters
	PostgreSQL	Double-quote (")
	SQL Server	Spaces or special characters
Allows quoted identifiers	SQL2003	Yes
	DB2	Yes
	MySQL	Yes
	Oracle	Yes
	PostgreSQL	Yes
	SQL Server	Yes
Quoted identifier symbol	SQL2003	Double-quote (")
	DB2	Double-quote (")
	MySQL	Quote (') or double-quote (") in ANSI compatibility mode

	Oracle	Double-quote (")
	PostgreSQL	Double-quote (")
	SQL Server	Double-quote (") or brackets ([]); brackets are preferred
Identifier may be reserved	SQL2003	No, unless as a quoted identifier
	DB2	Yes
	MySQL	No, unless as a quoted identifier
	Oracle	No, unless as a quoted identifier
	PostgreSQL	No, unless as a quoted identifier
	SQL Server	No, unless as a quoted identifier
Schema addressing	SQL2003	<i>Catalog.schema.object</i>
	DB2	Schema.object
	MySQL	Database.object
	Oracle	Schema.object
	PostgreSQL	Database.schema.object
	SQL Server	Server.database.schema.object
Identifier must be unique	SQL2003	Yes
	DB2	Yes
	MySQL	Yes
	Oracle	Yes
	PostgreSQL	Yes
	SQL Server	Yes
Other rules	SQL2003	None
	DB2	None
	MySQL	May not contain numbers only
	Oracle	Database links are limited to 128 bytes and may not be quoted identifiers
	PostgreSQL	None
	SQL Server	None

Identifiers must be unique within their scope. Thus, in our earlier discussion of the hierarchy of database objects, database names must be unique on a given instance of a database server, while the names of tables, views, functions, triggers, and stored procedures must be unique within a given schema. On the other hand, you *can* have a table and a stored procedure with the same name since they are different object types. The names of columns, keys, and indexes must be unique on a single table or view, and so forth. Check with the database platform documentation for more information — some platforms require unique identifiers where others may not. For example, DB2 requires all index identifiers to be unique throughout the database, while SQL Server requires that the index identifier be unique only for the table it depends on.

Remember, quoted identifiers (object names encapsulated within a special delimiter, usually double quotes) may be used to break some of the identifier rules specified earlier. Specifically, quoted identifiers may be used to bestow a reserved word as a name, or to allow normally unusable characters and symbols within a name. For example, you normally can't use the percent sign (%) in a table name. However, you can, if you must, use the percent sign in a table name so long as you always enclose that table name within double quotes. To name a table **expense%%ratios**, you would specify the name in quotes as "**expense%%ratios**". Again, remember that in SQL2003, such names are sometimes known as delimited identifiers.



Once you have created an object name as a quoted identifier, we recommend that you always reference it using its special delimiter.

2.2.2 Literals

SQL evaluates literal values as any explicit numeric, character string, temporal value (like a date or time), or Boolean value that is not an identifier or a keyword. SQL databases allow a variety of literal values in a SQL program. Literal values are allowed for most of the numeric, character, Boolean, and date datatypes. For example, SQL Server numeric datatypes include (among others) *INTEGER*, *REAL*, and *MONEY*. Thus, *numeric literals* can look like:

30

-117

+883.3338

-6.66

\$70000

2E5

7E-3

As the example illustrates, SQL Server allows signed or unsigned numerals, in scientific or normal notation. And since SQL Server has a money datatype, even a dollar sign can be included. SQL Server does not allow other symbols in numeric literals (besides 0 1 2 3 4 5 6 7 8 9 + - \$. E e), so do not include commas (or periods in Europe). Most databases interpret a comma in a numeric literal as a *list item separator*. Thus, the literal value 3,000 would be interpreted by the database as 3 and, separately, 000.

Boolean, character string, and date literals look like:

TRUE

'Hello world!'

'OCT-28-1966 22:14:30:00'

Character string literals should always be enclosed by single quotation marks (' '), the standard delimiter for all character string literals. Character string literals are not restricted just to the alphabet. In fact, any character in the character set can be represented as a string literal. All of the following are string literals:

'1998'

'70,000 + 14000'

'There once was a man from Nantucket,'

'Oct 28, 1966'

All of these examples are, in fact, compatible with the *CHARACTER* datatype. Remember not to confuse the string literal '1998' with the numeric literal 1998. Once string literals are associated with *CHARACTER* datatypes, it is poor practice to use them in arithmetic operations without explicitly converting them to a numeric datatype. Some database products will perform automatic conversion of string literals containing numbers when compared against any *DATE* or *NUMBER* datatype values.

By doubling the delimiter, you can effectively represent a single quotation mark in a literal string, if necessary. That is, use two quotation marks each time a single quotation mark is part of the value. This example taken from SQL Server illustrates the idea:

SELECT 'So he said "Who"s Le Petomaine?'"

This gives the result:

So he said 'Who's Le Petomaine?'

2.2.3 Operators

An operator is a symbol specifying an action that is performed on one or more expressions. Operators are used most often in *DELETE*, *INSERT*, *SELECT*, or *UPDATE* statements, but also are used frequently in the creation of database objects, such as stored procedures, functions, triggers, and views.

Operators typically fall into these categories:

Arithmetic operators

Supported by all databases

Assignment operators

Supported by all databases

Bitwise operators

Supported by Microsoft SQL Server

Comparison operators

Supported by all databases

Logical operators

Supported by DB2, Oracle, SQL Server, and PostgreSQL

Unary operators

Supported by DB2, Oracle, and SQL Server

2.2.3.1 Arithmetic operators

Arithmetic operators perform mathematical operations on two expressions of any datatypes in the numeric datatype category. See [Table 2-2](#) for a listing of the arithmetic operators.

Table 2-2. Arithmetic operators

Arithmetic operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modula (SQL Server only); returns the remainder of a division operation as an integer value



In DB2, Oracle, and SQL Server, the + and - operators also can be used to perform arithmetic operations on date values.

2.2.3.2 Assignment operators

Except in Oracle, which uses :=, the assignment operator (=) assigns a value to a variable or the alias of a column heading. In SQL Server, the keyword AS may serve as an operator for assigning table- or column-heading aliases.

2.2.3.3 Bitwise operators

Microsoft SQL Server provides bitwise operators as a shortcut to perform bit manipulations between two-integer expressions (see [Table 2-3](#)). Valid datatypes that are accessible to bitwise operators include *binary*, *bit*, *int*, *smallint*, *tinyint*, and *varbinary*.

Table 2-3. Bitwise operators

Bitwise operator	Meaning
&	Bitwise <i>AND</i> (two operands)
	Bitwise <i>OR</i> (two operands)
^	Bitwise exclusive <i>OR</i> (two operands)

2.2.3.4 Comparison operators

Comparison operators test whether two expressions are equal or unequal. The result of a comparison operation is a Boolean value: *TRUE*, *FALSE*, or *UNKNOWN*. Also, note that the ANSI standard behavior for a comparison operation where one or more of the expressions are NULL is NULL. For example, the expression *23 + NULL* returns NULL, as does the expression *Feb 23, 2002 + NULL*. See [Table 2-4](#) for a list of the comparison operators.

Table 2-4. Comparison operators

Comparison operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
!=	Not equal to (not ANSI standard)
!<	Not less than (not ANSI standard)
!>	Not greater than (not ANSI standard)

Boolean comparison operators are used most frequently in a *WHERE* clause to filter the rows that qualify for the search conditions. The following Microsoft SQL Server example uses the greater than or equal to comparison operation:

```
SELECT *  
  
FROM Products  
  
WHERE ProductID >= @MyProduct
```

2.2.3.5 Logical operators

Logical operators are commonly used in a *WHERE* clause to test for the truth of some condition. Logical operators return a Boolean value of either *TRUE* or *FALSE*. Logical operators also are discussed in [SELECT Statement](#). Not all database systems support all operators. See [Table 2-5](#) for a list of logical operators.

Table 2-5. Logical operators

Logical operator	Meaning
ALL	<i>TRUE</i> if all of a set of comparisons are <i>TRUE</i>
AND	<i>TRUE</i> if both Boolean expressions are <i>TRUE</i>
ANY	<i>TRUE</i> if any one of a set of comparisons is <i>TRUE</i>
BETWEEN	<i>TRUE</i> if the operand is within a range
EXISTS	<i>TRUE</i> if a subquery contains any rows
IN	<i>TRUE</i> if the operand is equal to one of a list of expressions or one or more rows returned by a subquery
LIKE	<i>TRUE</i> if the operand matches a pattern
NOT	Reverses the value of any other Boolean operator
OR	<i>TRUE</i> if either Boolean expression is <i>TRUE</i>
SOME	<i>TRUE</i> if some of a set of comparisons is <i>TRUE</i>

2.2.3.6 Unary operators

Unary operators perform an operation on only one expression of any of the datatypes of the numeric datatype category. Unary operators may be used on integer datatypes, though positive and negative may be used on any numeric datatype (see [Table 2-6](#)).

Table 2-6. Unary operators

Unary operator	Meaning
+	Numeric value is positive
-	Numeric value is negative
~	A bitwise NOT, returns the complement of the number (not in Oracle or DB2)

2.2.3.7 Operator precedence

Sometimes operator expressions become rather complex. When an expression has multiple operators, *operator precedence* determines the sequence in which the operations are performed. The order of execution can significantly affect the resulting value.

Operators have the precedence levels listed below. An operator on a higher level is evaluated before an operator on a lower level. The following listing denotes operators from highest to lowest precedence:

- () (parenthetical expressions)
- +, -, ~ (unary operators)
- *, /, % (mathematical operators)
- +, - (arithmetic operators)

- =, >, <, >=, <=, <>, !=, !>, !< (comparison operators)
- ^ (Bitwise Exclusive OR), & (Bitwise AND), | (Bitwise OR)
- NOT
- AND
- ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
- = (variable assignment)

Operators are evaluated from left to right when they are of equal precedence. However, parentheses are used to override the default precedence of the operators in an expression. Expressions within a parentheses are evaluated first, while operations outside the parentheses are evaluated next.

For example, the following expressions in an Oracle query return very different results:

```
SELECT 2 * 4 + 5 FROM dual
```

```
-- Evaluates to 8 + 5 which yields an expression result of 13.
```

```
SELECT 2 * (4 + 5) FROM dual
```

```
-- Evaluates to 2 * 9 which yields an expression result of 18.
```

In expressions with nested parentheses, the most deeply nested expression is evaluated first.

This next example contains nested parentheses, with the expression 5 - 3 in the most deeply nested set of parentheses. This expression yields a value of 2. Then the addition operator (+) adds this result to 4, which yields a value of 6. Finally, the 6 is multiplied by 2 to yield an expression result of 12:

```
SELECT 2 * (4 + (5 - 3) ) FROM dual
```

```
-- Evaluates to 2 * (4 + 2) which further evaluates to 2 * 6,
```

```
--and yields an expression result of 12.
```

```
RETURN
```

2.2.3.8 System delimiters and operators

String delimiters mark the boundaries of a string of alphanumeric characters. *System delimiters* are those symbols within the character set that have special significance to your database server. *Delimiters* are symbols that are used to judge the order or hierarchy of processes and list items. *Operators* are those delimiters used to judge values in comparison operations, including symbols commonly used for arithmetic or mathematical operations. [Table 2-7](#) lists the system delimiters and operators allowed by SQL.

Table 2-7. SQL delimiters and operators

Symbol	Usage	Example
+	Addition operator; in SQL Server, also serves as a concatenation operator	On all database platforms: <pre>SELECT MAX(emp_id) + 1 FROM employee</pre>
		As a subtraction operator: <pre>SELECT MIN(emp_id) - 1 FROM employee</pre>


-	Subtraction operator; also a range indicator in <i>CHECK</i> constraints	As a range operator, in a <i>CHECK</i> constraint: ALTER TABLE authors ADD CONSTRAINT authors_zip_num CHECK (zip LIKE '%[0-9]%)'
*	Multiplication operator	SELECT salary * 0.05 AS 'bonus' FROM employee;
/	Division operator	SELECT salary / 12 AS 'monthly' FROM employee;
=	Equality operator	SELECT * FROM employee WHERE lname = 'Fudd'
<>	Inequality operators (!= is a nonstandard equivalent on several platforms)	On all platforms: SELECT * FROM employee WHERE lname <> 'Fudd'
< <=	Less-than operator Less-than or equal-to operator	SELECT lname, emp_id, (salary * 0.05) AS bonus FROM employee WHERE (salary * 0.05) <= 10000 AND exempt_status < 3
> >=	Greater-than operator Greater-than or equal-to operator	SELECT lname, emp_id, (salary * 0.025) AS bonus FROM employee WHERE (salary * 0.025) > 10000 AND exempt_status >= 4
()	Expression, function calls, order of operations, and subquery delimiter	Expression: SELECT (salary / 12) AS monthly FROM employee WHERE exempt_status >= 4 Function call:

		<pre>SELECT SUM(travel_expenses) FROM "expense%%ratios"</pre>
		<pre>Order of operations: SELECT (salary / 12) AS monthly, ((salary / 12) / 2) AS biweekly FROM employee WHERE exempt_status >= 4</pre>
		<pre>Subquery: SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')</pre>
%	Wildcard attribute indicator (refer to LIKE Operator.)	<pre>SELECT * FROM employee WHERE lname LIKE 'Fud%'</pre>
,	List item separator	<pre>SELECT lname, fname, ssn, hire_date FROM employee WHERE lname = 'Fudd'</pre>
.	Identifier qualifier separator	<pre>SELECT * FROM scott.employee WHERE lname LIKE 'Fud%'</pre>
'	Character string indicators	<pre>SELECT * FROM employee WHERE lname LIKE 'FUD%' OR fname = 'ELMER'</pre>

"	Quoted identifier indicators	<pre>SELECT expense_date, SUM(travel_expense) FROM "expense%%ratios" WHERE expense_date BETWEEN '01-JAN-2004' AND '01-APR-2004'</pre>
—	Single-line comment delimiter (two dashes followed by a space)	<pre>-- Finds all employees like Fudd, -- Fudge, and Fudston SELECT * FROM employee WHERE Iname LIKE 'Fud%'</pre>
/* */	Beginning multiline comment delimiter Ending multiline comment indicator	<pre>/* Finds all employees like Fudd, Fudge, and Fudston */ SELECT * FROM employee WHERE Iname LIKE 'Fud%'</pre>

2.2.4 Keywords and Reserved Words

Just as certain symbols have special meaning and functionality within SQL, certain words and phrases have special significance. SQL *keywords* are words whose meanings are so closely tied to the operation of the RDBMS that they should not be used for any other purpose; generally, they are words used in a SQL statement. (Note that they *can* be used as an identifier on most platforms, but they shouldn't be.) For example, the word "SELECT" is a reserved word and should not be used as a table name.



It is generally a good idea to avoid naming columns or tables after a keyword that occurs in *any* major platform, because database applications are frequently converted from one platform to another.

Reserved words, on the other hand, do not have special significance now, but they probably will in a future release. To emphasize the fact that keywords should not be used as an identifier, but nevertheless could be, the SQL standard calls them "nonreserved keywords." Reserved words and keywords are not always words used in SQL statements, but may be words commonly associated with database technology. For example, *CASCADE* is used to describe data manipulations that allow their action, such as a delete or update, to "flow down," or cascade, to any subordinate tables. Reserved words and keywords are widely published so that programmers will not use them as identifiers that will, at some later revision, cause a problem.

SQL2003 specifies its own list of reserved words and keywords. In addition, the database platform specify their own list of reserved words and keywords because they each have their own extensions to the SQL command set. SQL standard keywords, as well as the keywords in the different vendor implementations, are shown in [Appendix B](#).

2.3 SQL2003 and Platform-Specific Datatypes

A table can contain one or many columns. Each column must be defined with a datatype that provides a general classification of the data that the column will store. In real-world applications, datatypes provide some control and efficiency as to how tables are defined and how the data is stored within the table. Using specific datatypes enables better, more understandable queries and helps control the integrity of data.

The tricky thing about SQL2003 datatypes is that they do not always map directly to an identical implementation in a given platform. Although the platforms specify "datatypes" that correspond to the SQL2003 datatypes, these are not always true SQL2003 datatypes. For example, MySQL's implementation of a *BIT* datatype is actually identical to a *CHAR(1)* datatype value. Nonetheless, each of the platform datatypes is close enough to the standard to be both easily understandable and job-ready.

The official SQL2003 datatypes (as opposed to platform-specific datatypes) fall into the general categories described in [Table 2-8](#). (Note that the SQL2003 standard contains a few rarely used datatypes (*ARRAY*, *MULTISET*, *REF*, and *ROW*) that are shown only in [Table 2-8](#) and not elsewhere in the book.)

Table 2-8. SQL2003 categories and datatypes

Category	Example datatypes and abbreviations	Description
BINARY	<i>BINARY LARGE OBJECT (BLOB)</i>	This datatype stores binary string values in hexadecimal format. Binary string values are stored without reference to any character set and without any length limit.
BOOLEAN	<i>BOOLEAN</i>	This datatype stores truth values—either <i>TRUE</i> or <i>FALSE</i> .
CHARACTER string types	<i>CHAR</i> <i>CHARACTER VARYING (VARCHAR)</i>	These datatypes can store any combination of characters from the applicable character set. The varying datatypes allow variable lengths, while the other datatypes allow only fixed lengths. Also, the variable-length datatypes automatically trim trailing spaces, while the other datatypes pad all open space.
	<i>NATIONAL CHARACTER (NCHAR)</i> <i>NATIONAL CHARACTER VARYING (NCHAR VARYING)</i>	The national character datatypes are designed to support a particular implementation-defined character set.
	<i>CHARACTER LARGE OBJECT (CLOB)</i>	<i>CHARACTER LARGE OBJECT</i> and <i>BINARY LARGE OBJECT</i> are collectively referred to as <i>large object string types</i> .
	<i>NATIONAL CHARACTER LARGE OBJECT (NCLOB)</i>	Same as <i>CHARACTER LARGE OBJECT</i> , but supports a particular implementation-defined character set.
DATALINK	<i>DATALINK</i>	Defines a reference to a file or other external data source that is not part of the SQL environment.
INTERVAL	<i>INTERVAL</i>	Specifies a set of time values or span of time.
COLLECTION	<i>ARRAY</i> <i>MULTISET</i>	<i>ARRAY</i> was offered in SQL99. <i>MULTISET</i> was added in SQL2003. Whereas an <i>ARRAY</i> is a set-length, ordered collection of elements, <i>MULTISET</i> is a variable-length, unordered collection of elements. The elements <i>ARRAY</i> and <i>MULTISET</i> must be of a predefined datatype.
NUMERIC	<i>INTEGER (INT)</i> <i>SMALLINT</i> <i>BIGINT</i> <i>NUMERIC(p,s)</i> <i>DEC[IMAL] (p,s)</i>	These datatypes store exact numeric values (integers or decimals) or approximate (floating point) values. <i>INT</i> , <i>BIGINT</i> , and <i>SMALLINT</i> store exact numeric values with a predefined precision and a scale of zero. <i>NUMERIC</i> and <i>DEC</i> store exact numeric values with a definable precision and a definable scale. <i>FLOAT</i> stores approximate numeric values with a definable precision, while <i>REAL</i> and <i>DOUBLE PRECISION</i> have predefined precisions. You may define a precision (p) and scale (s) for a <i>DECIMAL</i> , <i>FLOAT</i> , or <i>NUMERIC</i> datatype to indicate the total number of allowed digits in the number and the number of decimal places, respectively.

	<p><i>FLOAT(p,s)</i></p> <p><i>REAL</i></p> <p><i>DOUBLE PRECISION</i></p>	<p><i>INTEGER (INT)</i>, <i>SMALLINT</i>, and <i>DEC[IMAL]</i> (p,s) are sometimes referred to as <i>exact numeric types</i>, while <i>FLOAT (p,s)</i>, <i>REAL</i>, and <i>DOUBLE PRECISION</i> are sometimes called <i>approximate numeric types</i>.</p>
TEMPORAL	<p><i>DATE</i></p> <p><i>TIME</i></p> <p><i>TIME WITH TIME ZONE</i></p> <p><i>TIMESTAMP</i></p> <p><i>TIMESTAMP WITH TIME ZONE</i></p>	<p>These datatypes handle values related to time. <i>DATE</i> and <i>TIME</i> are self-explanatory. Datatypes with the <i>WITH TIME ZONE</i> suffix also include a time zone offset. The <i>TIMESTAMP</i> datatypes store a value that represents the a precise moment in time. Temporal types are also known as <i>datetime types</i>.</p>
XML	<i>XML</i>	<p>Stores XML data and can be used wherever a SQL datatype is allowed, such as a column of a table, field in a row, etc. Operations on the values of an XML type assume a tree-based internal data structure. The internal data structure is based on the XML Information Set Recommendation (Infoset) using a new document information item called the XML root information item.</p>

Not every database platform supports each and every ANSI SQL datatype. [Table 2-9](#) compares datatypes across the five platforms. The table is organized by datatype name. Be careful to look for footnotes when reading this table because some platforms support a datatype of a given name, but implement it in a different or contrary way than the ANSI standard and/or other vendors.


	<p>While platforms may support similarly named datatypes, their implementations can vary. Please consult the detailed information in this chapter for specific requirements of each platform's datatypes.</p>
--	---

Table 2-9. Comparison of platform-specific datatypes

Vendor datatype	DB2	MySQL	Oracle	PostgreSQL	SQL Server	SQL2003 datatype
<i>BFILE</i>			Y			none
<i>BIGINT</i>	Y	Y			Y	<i>BIGINT</i>
<i>BINARY</i>					Y	<i>BLOB</i>
<i>BINARY_FLOAT</i>			Y			<i>FLOAT</i>
<i>BINARY_DOUBLE</i>			Y			<i>DOUBLE PRECISION</i>
<i>BIT</i>		Y		Y	Y	none
<i>BIT VARYING, VARBIT</i>				Y		none
<i>BLOB</i>	Y	Y	Y			<i>BLOB</i>
<i>BOOL, BOOLEAN</i>		Y		Y		<i>BOOLEAN</i>
<i>BOX</i>				Y		none
<i>BYTEA</i>				Y		<i>BLOB</i>
<i>CHAR, CHARACTER</i>	Y	Y	Y	Y	Y	<i>CHARACTER</i>
<i>CHAR FOR BIT DATA</i>	Y					none
<i>CIDR</i>				Y		none
<i>CIRCLE</i>				Y		none

<i>CLOB</i>	Y		Y			<i>CLOB</i>
<i>CURSOR</i>					Y	none
<i>DATALINK</i>	Y					<i>DATALINK</i>
<i>DATE</i>	Y	Y	Y	Y		<i>DATE</i>
<i>DATETIME</i>		Y			Y	<i>TIMESTAMP</i>
<i>DBCLOB</i>	Y					<i>NCLOB</i>
<i>DEC, DECIMAL</i>	Y	Y	Y	Y	Y	<i>DECIMAL</i>
<i>DOUBLE, DOUBLE PRECISION</i>	Y	Y	Y	Y ^[1]	Y	<i>FLOAT</i>
<i>ENUM</i>		Y				none
<i>FLOAT</i>	Y	Y	Y		Y	<i>DOUBLE PRECISION</i>
<i>FLOAT4</i>				Y ^[2]		<i>FLOAT(P)</i>
<i>FLOAT8</i>				Y ^[3]		<i>FLOAT(P)</i>
<i>GRAPHIC</i>	Y					<i>BLOB</i>
<i>IMAGE</i>					Y	none
<i>INET</i>				Y		none
<i>INT, INTEGER</i>	Y	Y	Y	Y	Y	<i>INTEGER</i>
<i>INT2</i>				Y		<i>SMALLINT</i>
<i>INT4</i>				Y		<i>INT, INTEGER</i>
<i>INTERVAL</i>				Y		<i>INTERVAL</i>
<i>INTERVAL DAY TO SECOND</i>			Y			<i>INTERVAL DAY TO SECOND</i>
<i>INTERVAL YEAR TO MONTH</i>			Y			<i>INTERVAL YEAR TO MONTH</i>
<i>LINE</i>				Y		none
<i>LONG</i>			Y			none
<i>LONG VARCHAR</i>	Y					none
<i>LONGBLOB</i>		Y				<i>BINARY LARGE OBJECT</i>
<i>LONG RAW</i>			Y			<i>BLOB</i>
<i>LONG VARGRAPHIC</i>	Y					none
<i>LONGTEXT</i>		Y				none
<i>LSEG</i>				Y		none
<i>MACADDR</i>				Y		none
<i>MEDIUMTEXT</i>		Y				none
<i>MEDIUMBLOB</i>		Y				none
<i>MONEY</i>				Y ^[4]	Y	none
<i>NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, NCHAR VARYING, NVARCHAR</i>		Y	Y		Y	<i>NATIONAL CHARACTER VARYING</i>

<i>NCHAR, NATIONAL CHAR, NATIONAL CHARACTER</i>			Y		Y	<i>NATIONAL CHARACTER</i>
<i>NCLOB</i>			Y			<i>NCLOB</i>
<i>NTEXT, NATIONAL TEXT</i>					Y	<i>NCLOB</i>
<i>NVARCHAR2(N)</i>			Y			none
<i>NUMBER</i>	Y	Y	Y	Y ^[5]	Y	none
<i>NUMERIC</i>						<i>NUMERIC</i>
<i>OID</i>				Y		none
<i>PATH</i>				Y		none
<i>POINT</i>				Y		none
<i>POLYGON</i>				Y		none
<i>RAW</i>			Y			none
<i>REAL</i>	Y	Y	Y		Y	<i>REAL</i>
<i>ROWID</i>			Y			none
<i>ROWVERSION</i>					Y	none
<i>SERIAL, SERIAL4</i>				Y		none
<i>SERIAL8, BIGSERIAL</i>				Y		none
<i>SET</i>		Y				none
<i>SMALLDATETIME</i>					Y	none
<i>SMALLINT</i>	Y	Y	Y		Y	<i>SMALLINT</i>
<i>SMALLMONEY</i>					Y	none
<i>SQL_VARIANT</i>					Y	none
<i>TABLE</i>					Y	none
<i>TEXT</i>		Y		Y	Y	none
<i>TIME</i>	Y			Y		<i>TIME</i>
<i>TIMESPAN</i>				Y		<i>INTERVAL</i>
<i>TIMESTAMP</i>	Y		Y	Y	Y	<i>TIMESTAMP</i>
<i>TIMETZ</i>				Y		<i>TIME WITH TIME ZONE</i>
<i>TINYINT</i>					Y	none
<i>UNIQUEIDENTIFIER</i>					Y	none
<i>UROWID</i>			Y			none
<i>VARBINARY</i>					Y	<i>BLOB</i>
<i>VARCHAR, CHAR VARYING, CHARACTER VARYING</i>	Y	Y	Y ^[6]	Y	Y	<i>CHARACTER VARYING(N)</i>
<i>VARCHAR2</i>			Y			<i>CHARACTER VARYING</i>
<i>VARCHAR FOR BIT DATA</i>	Y					<i>BIT VARYING</i>
<i>VARGRAPHIC</i>	Y					<i>NCHAR VARYING</i>
<i>XMLTYPE</i>			Y			<i>XML</i>

[1] Synonym for *FLOAT*

[2] Synonym for *REAL*

[3] Synonym for *DOUBLE PRECISION*

[4] Synonym for *DECIMAL(9,2)*

[5] Synonym for *DECIMAL*

[6] Oracle vastly prefers *VARCHAR2*

Each of the following sections lists platform-specific datatypes, the SQL2003 datatype category (if any), and pertinent details. Descriptions are provided for non-SQL2003 datatypes.

2.3.1 DB2 Datatypes

As shown below, DB2 supports a rich variety of datatypes, including most of the SQL2003 datatypes:

BIGINT (SQL2003 Datatype: *BIGINT*)

Stores signed or unsigned integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.
Uses 8 bytes of storage.

BLOB (SQL2003 Datatype: *BLOB*)

Holds variable-length binary data up to 2,147,483,647 bytes in length.

CHAR(n), *CHARACTER(n)* (SQL2003 Datatype: *CHARACTER(n)*)

Holds fixed-length character data up to 254 bytes in length. Uses (*n*) bytes of storage.

CHAR(n) FOR BIT DATA (SQL2003 Datatype: *none*)

Holds a fixed number of values. Uses (*n*) bytes of storage.

CLOB (SQL2003 Datatype: *CLOB*)

Holds variable-length character data up to 2,147,483,647 bytes in length.

DATALINK(n) (SQL2003 Datatype: *DATALINK*)

Stores a link to a file external to the database. Uses $n+54$ bytes of storage.

DATE (SQL2003 Datatype: *DATE*)

Holds a calendar date without time of day. Uses 4 bytes of storage.

DBCLOB(n) (SQL2003 Datatype: *NCLOB*)

Holds variable-length double-byte character data up to 107,3741,823 characters in length.

DEC(p,s), *DECIMAL(p,s)* (SQL2003 Datatype: *DECIMAL*, *DECIMAL(p,s)*)

May have a precision of 1 to 31 and a scale of 0 to 31. Uses the integral part of $(p/2)+1$, where *p* is the

precision bytes storage.

DOUBLE, DOUBLE PRECISION (SQL2003 Datatype: *DOUBLE PRECISION*)

Holds floating point numbers of -1.79769E+308 through 1.79769E+308. Uses 8 bytes of storage.

FLOAT (SQL2003 Datatype: *FLOAT*)

A synonym of *DOUBLE PRECISION*.

FLOAT(p) (SQL2003 Datatype: *REAL, DOUBLE PRECISION*)

Can range from 1 through 53. If $p \leq 24$, then *FLOAT(p)* is a synonym for *REAL*. If $24 < p \leq 53$ then *FLOAT(p)* is a synonym for *DOUBLE PRECISION*.

GRAPHIC(n) (SQL2003 Datatype: *NATIONAL CHARACTER*)

Is not an image datatype, but rather holds fixed-length character strings (DBCS) up to 127 characters in length. Uses $n*2$ bytes of storage when using double-byte character sets or n bytes of storage when using single-byte character sets.

INT, INTEGER (SQL2003 Datatype: *INTEGER*)

Stores signed or unsigned integers between -2,147,483,648 and 2,147,483,647. Uses 4 bytes of storage.

LONG VARCHAR (SQL2003 Datatype: *VARCHAR*)

Holds variable-length character data up to 32,700 bytes in length. Uses 24 bytes of storage.

LONG VARCHAR FOR BIT DATA (SQL2003 Datatype: *BIT VARYING*)

Holds variable-length character data up to 32,700 bytes in length. Uses 24 bytes of storage.

LONG VARGRAPHIC (SQL2003 Datatype: none)

Holds variable-length double-byte character strings (DBCS) up to 16,350 characters in length. Uses 24 bytes of storage.

NUM(p,s), NUMERIC(p,s) (SQL2003 Datatype: *NUMERIC, NUMERIC(p,s)*)

Is a synonym for *DECIMAL*. Uses the integral part of $(p/2)+1$, where p is the precision bytes storage.

REAL (SQL2003 Datatype: *REAL*)

Holds floating point numbers of -3.402E+38 through 3.402E+38. Uses 4 bytes of storage.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

Stores signed or unsigned integers between -32,768 and 32,767. Uses 2 bytes of storage.

TIME (SQL2003 Datatype: *TIME*)

Holds the time of day. Uses 3 bytes of storage.

TIMESTAMP (SQL2003 Datatype: *TIMESTAMP*)

Stores the date and time. Uses 10 bytes of storage.

VARCHAR(n), *CHAR VARYING(n)*, *CHARACTER VARYING(n)* (SQL2003 Datatype: *CHARACTER VARYING*, *CHARACTER VARYING(n)*)

Holds variable-length character data up to 32,672 bytes in length. Uses $n+4$ bytes of storage.

VARCHAR(n) FOR BIT DATA (SQL2003 Datatype: *none*)

Stores a variable number of values. Uses $n+4$ bytes of storage.

VARGRAPHIC(n) (SQL2003 Datatype: *NCHAR VARYING*)

Holds variable-length double-byte character data up to 16,336 characters in length. Uses $(n*2)+4$ bytes of storage.

2.3.2 MySQL Datatypes

MySQL numeric datatypes support the following optional attributes:

UNSIGNED

The numeric value is assumed to be non-negative (positive or zero). For fixed-point datatypes like *DECIMAL* and *NUMERIC*, the space normally used to show a positive or negative condition of the numeric value can then be used as part of the value, providing a little extra numeric range in the column for *DECIMAL* and *NUMERIC* types. (There is no *SIGNED* optional attribute.)

ZEROFILL

Used for display formatting, this attribute tells MySQL that the numeric value is padded with zeros to its full size rather than spaces. *ZEROFILL* automatically forces the *UNSIGNED* attribute as well.

MySQL also enforces a maximum display size for columns of up to 255 characters. Columns longer than 255 characters are stored properly, but only 255 characters are displayed. Floating-point numeric datatypes may have a maximum of 30 digits after the decimal point.

The list below shows that MySQL supports most of the SQL2003 datatypes, plus several additional datatypes used to contain lists of values, as well as datatypes used for binary large objects (*BLOBs*). Datatypes that extend the ANSI standard include *TEXT*, *ENUM*, *SET*, and *MEDIUMINT*. Special datatype attributes that go beyond the ANSI standard include *AUTO_INCREMENT*, *BINARY*, *NULL*, *UNSIGNED*, and *ZEROFILL*.

BIGINT(n) [*UNSIGNED*] [*ZEROFILL*] (SQL2003 Datatype: *BIGINT*)

Stores signed or unsigned integers. The signed range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The unsigned range is 0 to 18,446,744,073,709,551,615. *BIGINT* can perform imprecise calculations due to rounding issues.

BIT, *BOOL* (SQL2003 Datatype: *none*)

Synonyms for *TINYINT*.

BLOB (SQL2003 Datatype: *BLOB*)

Stores up to 65,535 characters of data. Support for indexing *BLOB* columns is found only in MySQL Version 3.23.2 or greater (a feature not found in any other platform covered in this book). In MySQL, *BLOBs* are functionally equivalent to the MySQL datatype *VARCHAR BINARY* (discussed below) with the default upper limit. *BLOBs* always require case-sensitive comparisons. They differ from a MySQL *VARCHAR BINARY* column by not allowing *DEFAULT* values and not removing trailing spaces. Do not perform *GROUP BY* or *ORDER BY* on *BLOB* columns. They also are stored separately from their table, whereas all other datatypes in MySQL are stored in the table file structure itself.

CHAR(n)[BINARY], *CHARACTER(n) [BINARY]* (SQL2003 Datatype: *CHARACTER(n)*)

Contains a fixed-length character string of 1 to 255 characters in length. *CHAR* pads with blank spaces when it

stores values, but trims spaces upon retrieval as ANSI SQL2003 *VARCHAR* does. The *BINARY* option allows binary searches rather than dictionary-order, case-insensitive searches.

DATE (SQL2003 Datatype: DATE)

Stores a date from 1000-01-01 to 9999-12-31 (delimited by quotes). MySQL displays these values by default as YYYY-MM-DD, though the user may specify some other display format.

DATETIME (SQL2003 Datatype: TIMESTAMP)

Stores date and time values within the range of 1000-01-01 00:00:00 to 9999-12-31 23:59:59.

DECIMAL [(p[,s])] [ZEROFILL] (SQL2003 Datatype: DECIMAL(PRECISION, SCALE))

Stores exact numeric values as if they were strings, using a single character for each digit. Precision is 10 if omitted, and scale is 0 if omitted.

DOUBLE[(p,s)] [ZEROFILL], DOUBLE PRECISION[(p,s)] [ZEROFILL] (SQL2003 Datatype: DOUBLE PRECISION)

Holds double-precision numeric values and is otherwise identical to the double-precision *FLOAT* datatype, except that its allowable range is -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308.

ENUM ("val1""val2," . . . n) (SQL2003 Datatype: none)

A datatype whose value must be one of those contained in the list of values (expressed as strings but stored as integers), NULL, or an empty string("") as an error value. Up to 65,535 distinct values are allowed.

FLOAT[(p)] [ZEROFILL], float(p,s) [ZEROFILL] (SQL2003 Datatype: FLOAT(P))

Stores floating-point numbers in the range -3.402823466E+38 to -1.175494351E-38 and 1.175494351E-38 to 3.402823466E+38. *FLOAT* without a precision, or with a precision of ≤ 24 , is single precision. Otherwise, *FLOAT* is double precision. When specified alone, precision can range from 0 to 53. When you specify both precision and scale, precision may be as high as 255 and the scale may be as high as 253. All *FLOAT* calculations in MySQL are done with double precision and may, since *FLOAT* is an approximate datatype, encounter rounding errors.

INT[EGER] [(n)] [UNSIGNED] [ZEROFILL] [AUTO_INCREMENT] (SQL2003 Datatype: INT, INTEGER)

Stores signed or unsigned integers within the range of -2,147,483,648 to 2,147,483,647 on ISAM tables. Unsigned, the range is 0 to 4,294,967,295 on ISAM tables. The range of values varies slightly on other types of tables. *AUTO_INCREMENT* is available to all of the *INT* variants. It creates a unique row identity for all new rows added to the table. (Refer to [CREATE/ALTER TABLE Statement](#) for more information on *AUTO_INCREMENT*.)

LONGBLOB (SQL2003 Datatype: BINARY LARGE OBJECT)

Stores *BLOB* data up to 4,294,967,295 characters in length. Note that this might be too much information for some client/server protocols to support.

LONGTEXT (SQL2003 Datatype: CLOB)

Stores *TEXT* data up to 4,294,967,295 characters in length. Note that this might be too much data for some client/server protocols to support.

MEDIUMBLOB (SQL2003 Datatype: none)

Stores *BLOB* data up to 16,777,215 characters in length.

MEDIUMTEXT (SQL2003 Datatype: none)

Stores *TEXT* data up to 16,777,215 characters in length.

MEDIUMINT[(*n*)] [*UNSIGNED*] [*ZEROFILL*] (*SQL2003 Datatype: none*)

Stores signed or unsigned integers within the range of 8,388,608 to -8,388,608. The unsigned range is 0 to 16,777,215.

NCHAR(*n*) [*BINARY*], [*NATIONAL*] *CHAR*(*n*) [*BINARY*] (*SQL2003 Datatype: NCHAR(n)*)

Synonyms for *CHAR*. The *NCHAR* datatypes provide UNICODE support beginning in MySQL v4.1.

NUMERIC(*p,s*) (*SQL2003 Datatype: DECIMAL(p,s)*)

A synonym of *DECIMAL*.

NVARCHAR(*n*) [*BINARY*], [*national*] *VARCHAR*(*n*) [*BINARY*], *NATIONAL CHARACTER VARYING*(*n*) [*BINARY*] (*SQL2003 Datatype: NCHAR VARYING*)

Synonyms for *VARYING [BINARY]*. Holds variable length character strings up to 255 characters in length. Values are stored and compared as case-insensitive unless the *BINARY* keyword is used.

REAL(*p,s*) (*SQL2003 Datatype: REAL*)

Is a synonym of *DOUBLE PRECISION*.

SET("val1," "val2," . . . *n*) (*SQL2003 Datatype: none*)

Is a *CHAR* datatype whose value must be equal to zero or more values specified in the list of values. Up to 64 items are allowed in the list of values.

SMALLINT[(*n*)] [*UNSIGNED*] [*ZEROFILL*] (*SQL2003 Datatype: SMALLINT*)

Stores signed or unsigned integers. The signed range is from -32768 to 32,767. The unsigned range is 0 to 65,535.

TEXT (*SQL2003 Datatype: none*)

Stores up to 65,535 characters of data. *TEXT* datatypes are stored separately from their tables, whereas all other datatypes are stored in their respective table file structures. *TEXT* is functionally equivalent to *VARCHAR* with no specific upper limit (besides the maximum size of the column), and requires case-insensitive comparisons. *TEXT* differs from a standard *VARCHAR* column by not allowing *DEFAULT* values, and by not removing trailing spaces. *TEXT* columns cannot be used in *GROUP BY* or *ORDER BY* clauses. In addition, support for indexing *TEXT* columns comes only in MySQL Version 3.23.2 or greater.

2.3.3 Oracle Datatypes

As shown below, Oracle supports a rich variety of datatypes, including most of the SQL2003 datatypes.

BFILE (*SQL2003 Datatype: DATALINK*)

Holds a pointer to a *BLOB* stored outside the database, but present on the local server, of up to 4 GB in size. The database streams input (but not output) access to the external *BLOB*. If you delete a row containing a *BFILE* value, only the pointer value is deleted. The actual file structure is not deleted.

BINARY_FLOAT (*SQL2003 Datatype: FLOAT*)

Holds a 32-bit floating point number.

BINARY_DOUBLE (*SQL2003 Datatype: FLOAT*)

Holds a 64-bit floating point number.

BLOB (SQL2003 Datatype: *BLOB*)

Holds a binary large object (*BLOB*) value of between 8 and 128 terabytes in size, depending on the database blocksize. In Oracle, large binary objects (*BLOBs*, *CLOBs*, and *NCLOBs*) have the following restrictions:

- They cannot be selected remotely.
- They cannot be stored in clusters.
- They cannot compose a varray.
- They cannot be a component of an *ORDER BY* or *GROUP BY* clause in a query.
- They cannot be used by an aggregate function in a query.
- They cannot be referenced in queries using *DISTINCT*, *UNIQUE*, or joins.
- They cannot be referenced in *ANALYZE...COMPUTE* or *ANALYZE...ESTIMATE* statements.
- They cannot be part of a primary key or index key.
- They cannot be used in the *UPDATE OF* clause in an *UPDATE* trigger.

CHAR(n)[BYTE | CHAR], *CHARACTER(n)[BYTE | CHAR]* (SQL2003 Datatype: *CHARACTER(n)*)

Holds fixed-length character data up to 2,000 bytes in length. *BYTE* tells Oracle to use bytes for the size measurement. *CHAR* tells Oracle to use characters for the size measurement.

CLOB (SQL2003 Datatype: *CLOB*)

Stores a character large object (*CLOB*) value of between 8 and 128 terabytes in size, depending on the database blocksize. See *BLOB* for a list of restrictions on the use of the *CLOB* type.

DATE (SQL2003 Datatype: *DATE*)

Stores a valid date and time within the range of 4712BC-01-01 00:00:00 to 9999AD-12-31 23:59:59.

DECIMAL(p,s) (SQL2003 Datatype: *DECIMAL(p,s)*)

Is a synonym of *NUMBER* that accepts precision and scale arguments.

DOUBLE PRECISION (SQL2003 Datatype: *DOUBLE PRECISION*)

Stores floating point values with double precision, the same as *FLOAT(126)*.

FLOAT(n) (SQL2003 Datatype: *FLOAT(n)*)

Stores floating-point numeric values with a binary precision of up to 126.

INTEGER(n) (SQL2003 Datatype: *INTEGER*)

Stores signed and unsigned integer values with a precision of up to 38. *INTEGER* is treated as a synonym for *NUMBER*.

INTERVAL DAY (n) TO SECOND (x) (SQL2003 Datatype: *INTERVAL*)

Stores a time span in days, hours, minutes, and seconds where *n* is the number of digits in the day field (0 to 9 acceptable, 2 is the default) and *x* is the number of digits used for fractional seconds in the seconds field (0 to 9 acceptable, 6 is the default).

INTERVAL YEAR (n) TO MONTH (SQL2003 Datatype: *INTERVAL*)

Stores a time span in years and months where *n* is the number of digits in the year field. The value *n* can be 0

to 9, with a default of 2.

LONG (SQL2003 Datatype: none)

Stores variable-length character data up to 2 gigabytes in size. Please note, however, that *LONG* is not scheduled for long-term support by Oracle. Use another datatype, such as *CLOB*, instead of *LONG*, whenever possible.

LONG RAW (SQL2003 Datatype: none)

Stores raw variable-length binary data up to 2 gigabytes in size. *LONG RAW* and *RAW* are typically used to store graphics, sounds, documents, and other large data structures. *BLOB* is preferred over *LONGRAW* in Oracle because *BLOB* has fewer restrictions on its use. *LONGRAW* is also deprecated.

NATIONAL CHARACTER VARYING (n), *NATIONAL CHAR VARYING (n)*,

NCHAR VARYING (n) (SQL2003 Datatype: *NCHAR VARYING (n)*)

The same as *NVARCHAR2*.

NCHAR(n), *NATIONAL CHARACTER(n)*, *NATIONAL CHAR(n)* (SQL2003 Datatype: *NATIONAL CHARACTER*)

Holds UNICODE character data of 1 to 2,000 bytes in length. Default size is 1 byte.

NCLOB (SQL2003 Datatype: *NCLOB*)

Represents a *CLOB* that supports multibyte and UNICODE values of between 8 and 128 terabytes in size, depending on the database blocksize. See *BLOB* for a list of restrictions on the use of the *CLOB* type.

NUMBER (p,s), *NUMERIC (p,s)* (SQL2003 Datatype: *NUMERIC (p,s)*)

Stores a number with a precision of 1 to 38 and a scale of -84 to 127.

NVARCHAR2(n) (SQL2003 Datatype: none)

Represents Oracle's preferred UNICODE variable-length character datatype. It can hold 1 to 4,000 bytes.

RAW(n) (SQL2003 Datatype: none)

Stores raw, variable-length binary data up to 2,000 bytes in size. The value *n* is the specified size of the datatype. *RAW* is also deprecated in Oracle 10g. See *LONG RAW*.

REAL (SQL2003 Datatype: *REAL*)

Stores floating-point values as single-precision and is the same as *FLOAT(63)*.

ROWID (SQL2003 Datatype: none)

Represents a unique, base-64 identifier for each row in a table, often used in conjunction with the *ROWID* pseudocolumn.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

The same as *INTEGER*.

TIMESTAMP(n){[WITH TIME ZONE] | [WITH LOCAL TIME ZONE]} (SQL2003 Datatype: *TIMESTAMP[WITH TIME ZONE]*)

A full date and time value where *n* is the number of digits (0 to 9 are acceptable, 6 is the default) in the fractional part of the seconds field. *WITH TIME ZONE* stores whatever time zone you pass to it (the default is your session time zone) and returns a time value in that same time zone. *WITH LOCAL TIME ZONE* stores data in the time zone of the current session and returns data in the time zone of the user's session.

UROWID [(n)] (SQL2003 Datatype: none)

Stores a base-64 value showing the logical address of the row in its table. It defaults to 4,000 bytes in size, but you may optionally specify its size up to 4,000 bytes.

VARCHAR(n), *CHARACTER VARYING(n)*, *CHAR VARYING(n)* (SQL2003 Datatype: *CHARACTER VARYING(n)*)

Holds variable-length character data of 1 to 4,000 bytes in size.



Oracle does not recommend using *VARCHAR*. Oracle has encouraged the use of *VARCHAR2* instead for many years.

VARCHAR2(n [BYTE | CHAR]) (SQL2003 Datatype: *CHARACTER VARYING(n)*)

Holds variable-length character data up to 4,000 bytes in length as defined by *n*. *BYTE* tells Oracle to use bytes for the size measurement. *CHAR* tells Oracle to use characters for the size measurement. If you use *CHAR*, Oracle internally must still transform that into some number of bytes, which is then subject to the 4,000-byte upper limit.

XMLTYPE (SQL2003 Datatype: *XML*)

Stores XML data within the Oracle database. The XML data is accessed using XPath expressions as well as a number of built-in XPath functions, SQL functions, and PL/SQL packages. The *XMLTYPE* datatype is a system-defined type, so it is usable as an argument in functions, or as a datatype of column in a table or view. When used in a table, the data can be stored in a *CLOB* column or object-rationally.

2.3.4 PostgreSQL Datatypes

PostgreSQL database supports most SQL2003 datatypes, plus an extremely rich set of datatypes that store spatial and geometric data. PostgreSQL sports a rich set of operators and functions especially for the geometric datatypes, including capabilities such as rotation, finding intersections, and scaling. PostgreSQL also supports additional versions of existing datatypes that are smaller and take up less disk space than their corresponding primary datatypes. For example, PostgreSQL offers several variations on *INTEGER* to accommodate small or large numbers and thereby consume proportionally less or more space.

BIGSERIAL

See *SERIALS*.

BIT (SQL2003 Datatype: *BIT*)

A fixed-length bit string.

BIT VARYING(n), *varbit(n)* (SQL2003 Datatype: *BIT VARYING*)

Denotes a variable-length bit string *n* bits in length.

BOOL, *BOOLEAN* (SQL2003 Datatype: *BOOLEAN*)

Stores a logical Boolean (true/false/unknown) value. The keywords *TRUE* and *FALSE* are preferred, but PostgreSQL supports the following valid literal values for the "true" state: *TRUE*, *t*, *true*, *y*, *yes*, and *1*. Valid "false" values are: *FALSE*, *f*, *false*, *n*, *no*, and *0*.

BOX (*x1*, *y1*), (*x2*, *y2*)) (SQL2003 Datatype: none)

Stores the values of a rectangular box in a 2D plane. Values are stored in 32 bytes and are represented as ((*x1*,*y1*),(*x2*,*y2*)), signifying the opposite corners of the box (upper-right and lower-left corners, respectively). The outer parentheses are optional.

BYTEA (SQL2003 Datatype: *BINARY LARGE OBJECT*)

Raw, binary data such as that used to store graphics, sound, or documents. For storage, this datatype requires 4 bytes plus the actual size of the binary string.

CHAR(*n*), *CHARACTER*(*n*) (SQL2003 Datatype: *CHARACTER*(*n*))

Contains a fixed-length character string padded with spaces up to a length of *n*. Attempting to insert a value longer than *n* results in an error (unless the extra length is composed of spaces, which are then truncated such that the result fits in *n* characters).

CIDR (*x.x.x.x/y*) (SQL2003 Datatype: none)

Describes an IP-Version 4 network or host address in a 12-byte storage space. The range is any valid IPv4 network address. Data in *CIDR* datatypes is represented as *x.x.x.x/y* where the *xs* are the IP address and *y* is the number of bits in the netmask. *CIDR* does not accept nonzero bits to the right of a zero bit in the netmask.

CIRCLE *x*, *y*, *r* (SQL2003 Datatype: none)

Describes a circle in a 2D plane. Values are stored in 24 bytes of storage space and are represented as: *x*, *y*, *r*. The *x*, *y* value represents the coordinates of the center of the circle, while *r* represents the length of the radius. Parentheses or arrow brackets may optionally delimit the values for *x*, *y*, and *r*.

DATE (SQL2003 Datatype: *DATE*)

Holds a calendar date (year, day, and month) without the time of day in a 4-byte storage space. Dates must be between 4713 BC and 32767 AD. *DATE*'s lowest resolution, naturally, is to the day.

DATETIME (SQL2003 Datatype: *TIMESTAMP*)

Holds a calendar date with a specific time of day.

DECIMAL [(*p*,*s*)], *NUMERIC*[(*p*,*s*)](SQL2003 Datatype: *DECIMAL* (*PRECISION*, *SCALE*), *NUMERIC* (*p*,*s*))

Stores exact numeric values with a precision (*p*) as high as 9 and a scale (*s*) of 0, with no upper limit.

FLOAT4, *REAL* (SQL2003 Datatype: *FLOAT*(*p*))

Stores floating-point numbers with a precision of 8 or less and 6 decimal places.

FLOAT8, *DOUBLE PRECISION* (SQL2003 Datatype: *FLOAT*(*p*), 7 <= *p* < 16)

Stores floating-point numbers with a precision of 16 or less, and 15 decimal places.

INET (*x.x.x.x/y*)

Stores an IP-Version 4 network or host address in a 12-byte storage space. The range is any valid IPv4 network address. The *x*'s represent the IP address, and *y* is the number of bits in the netmask. The netmask defaults to 32. Unlike *CIDR*, *INET* accepts nonzero bits to the right of the netmask.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

Stores signed or unsigned 2-byte integers within a range of -32,768 to 32,767. *INT2* is a synonym.

INTEGER (SQL2003 Datatype: *INTEGER*)

Stores signed or unsigned 4-byte integers within a range of -2,147,483,648 to 2,147,483,647. *INT4* is a synonym.

INT8 (SQL2003 Datatype: none)

Stores signed or unsigned 8-byte integers with a range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

INTERVAL(p) (SQL2003 Datatype: none)

Holds general-use time-span values within the range of -178,000,000 to 178,000,000 years in a 12-byte storage space. *INTERVAL*'s lowest resolution is to the microsecond. This is a different datatype than the ANSI standard, which requires an interval qualifier such as *INTERVAL YEAR TO MONTH*.

LINE ((*x1*, *y1*), (*x2*, *y2*)) (SQL2003 Datatype: none)

Holds line data, without endpoints, in 2D plane values. Values are stored in 32 bytes and are represented as ((*x1*,*y1*), (*x2*,*y2*)), indicating the start and end points of a line. Parentheses are optional for line syntax.

LSEG ((*x1*, *y1*), (*x2*, *y2*)) (SQL2003 Datatype: none)

Holds line segment (*LSEG*) data, with endpoints, in a 2D plane. Values are stored in 32 bytes and are represented as ((*x1*,*y1*), (*x2*,*y2*)). Parentheses are optional for *LSEG* syntax. To those who are interested, the "line segment" is what most people traditionally think of as a line. For example, the lines on a playing field are actually line segments.



In true geometric nomenclature, a *line* stretches to infinity, having no terminus at either end, while a *line segment* has end points. PostgreSQL allows datatypes for both, but they are functionally equivalent.

MACADDR (SQL2003 Datatype: none)

Holds a value for the *MAC* address of a computer's network interface card in a 6-byte storage space. *MACADDR* accepts a number of industry standard representations, such as:

08002B:010203
08002B-010203
0800.2B01.0203
08-00-2B-01-02-03
08:00:2B:01:02:03

MONEY, *DECIMAL(9,2)* (SQL2003 Datatype: none)

Stores U.S.-style currency values in the range of (-21,474,836.48 to 21,474,836.47).

NUMERIC [(*p*, *s*)], *DECIMAL* [(*p*, *s*)] (SQL2003 Datatype: none)

Stores exact numeric values with a precision (*p*) and scale (*s*).

OID (SQL2003 Datatype: none)

Stores unique object identifiers.

PATH ((*x1*, *y1*), ...*n*), *Path* [(*x1*, *y1*), ...*n*] (SQL2003 Datatype: none)

Describes an open and closed geometric path in a 2D plane. Values are represented as $[(x1, y1), \dots]$ and consume $4 + 32n$ bytes of storage space. Each (x,y) value represents a point on the path. Paths are either open, where the first and last points do not intersect, or closed, where the first and last points do intersect. Parentheses are used to encapsulate closed paths, while brackets encapsulate open paths.

POINT (x, y) (SQL2003 Datatype: none)

Stores values for a geometric point in a 2D plane in a 16-byte storage space. Values are represented as (x,y) . The point is the basis for all other two-dimensional spatial datatypes supported in PostgreSQL. Parentheses are optional for point syntax.

POLYGON ($(x1, y1), \dots n$) (SQL2003 Datatype: none)

Stores values for a closed geometric path in a 2D plane using $4 + 32n$ bytes of storage. Values are represented as $[(x1,y1), \dots]$. *POLYGON* is essentially a closed path datatype.

SERIAL, *SERIAL4* (SQL2003 Datatype: none)

Stores an autoincrementing, unique, integer ID for indexing and cross-referencing. These types store up to 4 bytes of data (a range of numbers from 1 to 2,147,483,647). Tables defined with this datatype cannot be directly dropped. You must first issue the *DROP SEQUENCE* command, then follow up with the *DROP TABLE* command.

SERIAL8, *BIGSERIAL* (SQL2003 Datatype: none)

Stores an autoincrementing, unique integer ID for indexing and cross-referencing. It stores up to 8 bytes of data (a range of numbers from 1 to 9,223,372,036,854,775,807). Tables defined with this datatype cannot be directly dropped. You must first issue the *DROP SEQUENCE* command, and then follow up with the *DROP TABLE* command.

TEXT (SQL2003 Datatype: *CLOB*)

Stores large, variable-length, character-string data up to 1 gigabyte. PostgreSQL automatically compresses *TEXT* strings, so the disk size may be less than the string size.

TIME [(p)] [*WITHOUT TIME ZONE* | *WITH TIME ZONE*] (SQL2003 Datatype: *TIME*)

Holds the time of day and stores either no time zone (using 8 bytes of storage space) or the time zone of the database server (using 12 bytes of storage space). The allowable range is from 00:00:00.00 to 23:59:59.99. The lowest granularity is 1 microsecond. Note that time zone information on most Unix systems is available only for the years 1902 through 2038.

TIMESPAN (SQL2003 Datatype: none)

Holds a value that represents a specific span of time. The ANSI datatype most like PostgreSQL's *TIMESPAN* is *INTERVAL*.

TIMESTAMP [(p)] [*WITHOUT TIME ZONE* | *WITH TIME ZONE*] (SQL2003 Datatype: *TIMESTAMP* [*WITH TIME ZONE* | *WITHOUT TIME ZONE*])

Stores the date and time and stores either no time zone or the time zone of the database server. The range of values is from 4713 BC to 1465001 AD. *TIMESTAMP* uses 8 bytes of storage space per value. The lowest granularity is 1 microsecond. Note that time zone information on most Unix systems is available only for the years 1902 through 2038.

TIMETZ (SQL2003 Datatype: *TIME WITH TIME ZONE*)

Holds the time of day, including the time zone.

VARCHAR(n), *CHARACTER VARYING*(n) (SQL2003 Datatype: *CHARACTER VARYING*(n))

Stores variable-length character strings up to a length of n . Trailing spaces are not stored.

2.3.5 SQL Server Datatypes

The list below shows that Microsoft SQL Server supports most SQL2003 datatypes. SQL Server supports additional datatypes used to uniquely identify rows of data within a table and across multiple servers, such as *UNIQUEIDENTIFIER*, in support of Microsoft's hardware philosophy of "scale-out" (that is, deploying on many Intel-based servers), rather than "scale-up" (deploying on a single, huge, high-end Unix server or a Windows Data Center Server).



An interesting side note about SQL Server dates: SQL Server supports dates starting at the year 1753. You can't store dates prior to that year using any of SQL Server's date datatypes. Why not? The rationale is that the English-speaking world started using the Gregorian calendar in 1753 (the Julian calendar was used prior to September, 1753) and converting dates prior to Julian to the Gregorian calendar can be quite challenging.

BIGINT (SQL2003 Datatype: *BIGINT*)

Stores signed and unsigned integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 using 8 bytes of storage space. See *INT* for *IDENTITY* property rules that also apply to *BIGINT*.

BINARY[(*n*)] (SQL2003 Datatype: *BLOB*)

Stores a fixed-length binary value of 1 to 8,000 bytes in size. *BINARY* datatypes consume $n + 4$ bytes of storage space.

BIT (SQL2003 Datatype: *BOOLEAN*)

Stores 1, 0, or NULL to indicate "unknown." Up to 8 *BIT* columns on a single table will be stored in a single byte. An additional 8 *BIT* columns consume one more byte of storage space. *BIT* columns cannot be indexed.

CHAR[(*n*)], *CHARACTER*[(*n*)] (SQL2003 Datatype: *CHARACTER*(*n*))

Holds fixed-length character data of 1 to 8,000 characters in length. Any unused space is, by default, padded with spaces. (You can disable the automatic padding of spaces.) Storage size is n bytes.

CURSOR (SQL2003 Datatype: none)

A special datatype used to describe a cursor as a variable or stored procedure *OUTPUT* parameter. It cannot be used in a *CREATE TABLE* statement. The *CURSOR* datatype is always nullable.

DATETIME (SQL2003 Datatype: *TIMESTAMP*)

Holds date and time data within the range of 1753-01-01 00:00:00 through 9999-12-31 23:59:59. Values are stored in an 8-byte storage space.

DECIMAL (*p,s*), *DEC* (*p,s*), *NUMERIC* (*p,s*) (SQL2003 Datatype: *DECIMAL* (*p,s*), *NUMERIC* (*p,s*))

Stores decimal values up to 38 digits long. The values p and s define precision and scale, respectively. The default value for scale is 0. The precision of the datatype determines how much storage space it will consume:

- Precision 1-9 uses 5 bytes
- Precision 10-19 uses 9 bytes
- Precision 20-28 uses 13 bytes
- Precision 29-39 uses 17 bytes

See *INT* for *IDENTITY* property rules that also apply to *DECIMAL*.

DOUBLE PRECISION (SQL2003 Datatype: none)

A synonym for *FLOAT*(53).

FLOAT [(*n*)] (SQL2003 Datatype: *FLOAT*, *FLOAT* (*n*))

Holds floating-point numbers in the range -1.79E+308 through 1.79E+308. Precision, represented by *n*, may be 1 to 53. Storage size is 4 bytes for 7 digits, where *n* is 1 to 24. Anything larger requires 8 bytes of storage.

IMAGE (SQL2003 Datatype: *BLOB*)

Stores a variable-length binary value up to 2,147,483,647 bytes in length. This datatype is commonly used to store graphics, sounds, and files like MS-Word documents and MS-Excel spreadsheets. *IMAGE* cannot be freely manipulated. *IMAGE* and *TEXT* columns have a lot of constraints on how they can be used. See *TEXT* for a list of the commands and functions that work on an *IMAGE* datatype.

INT [*IDENTITY* [(*seed*, *increment*)]] (SQL2003 Datatype: *INTEGER*)

Stores signed or unsigned integers between -2,147,483,648 and 2,147,483,647 in 4 bytes of storage space. All integer datatypes, as well as the decimal type, support the *IDENTITY* property. An identity is an automatically incrementing row identifier. Refer to the [CREATE/ALTER TABLE Statement](#).

MONEY (SQL2003 Datatype: none)

Stores monetary values in a range from -922,337,203,685,477.5808 to 922,337,203,685,477.5807. Values are stored in an 8-byte storage space.

NCHAR(*n*), *NATIONAL CHAR*(*n*), *NATIONAL CHARACTER*(*n*) (SQL2003 Datatype: *NATIONAL CHARACTER* (*n*))

Holds fixed-length UNICODE data up to 4,000 characters in length. *NCHAR* consumes twice *n* in storage space.

NTEXT, *NATIONAL TEXT* (SQL2003 Datatype: *NCLOB*)

Holds UNICODE text passages up to 1,073,741,823 characters in length. See *TEXT* for rules about the commands and functions available for *NTEXT*.

NUMERIC(*p,s*) (SQL2003 Datatype: *DECIMAL*(*p,s*))

A synonym for *DECIMAL*. See *INT* for rules about the *IDENTITY* property.

NVARCHAR(*n*), *NATIONAL CHAR VARYING*(*n*), *NATIONAL CHARACTER VARYING*(*n*) (SQL2003 Datatype: *NATIONAL CHARACTER VARYING*(*n*))

Holds variable-length UNICODE data up to 4,000 characters in length. Storage space consumed is double the character length (characters * 2) inserted into the field. The system setting *SET ANSI_PADDING* is always enabled (ON) for *NCHAR* and *NVARCHAR* fields in SQL Server.

REAL, *FLOAT*(24) (SQL2003 Datatype: *REAL*)

Holds floating point numbers of -3.40E + 38 through 3.40 E+38 in a 4-byte storage space. *REAL* is functionally equivalent to *FLOAT*(24).

ROWVERSION (SQL2003 Datatype: none)

A unique number within a database that is updated whenever a row is updated. Called *TIMESTAMP* in earlier versions.

SMALLDATETIME (SQL2003 Datatype: none)

Holds date and time data within the range of `1900-01-01 00:00' through `2079-06-06 23:59' down to the nearest minute. (Minutes are rounded down when seconds are 29.998 or less. Otherwise, they are rounded up.) Values are stored in a 4-byte storage space.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

Stores signed or unsigned integers between -32,768 and 32,767 in 2 bytes of storage space. See *INT* for rules about the *IDENTITY* property, which applies to this type.

SMALLMONEY (SQL2003 Datatype: none)

Stores monetary values within the range of 214,748.3648 and -214,748.3647. Values are stored in a 4-byte storage space.

SQL_VARIANT (SQL2003 Datatype: none)

Stores values of other SQL Server-supported datatypes, except *TEXT*, *NTEXT*, *ROWVERSION*, and other *SQL_VARIANT*s. It can store up to 8,016 bytes of data and supports NULL and DEFAULT values. *SQL_VARIANT* is used in columns, parameters, variables, and return values of functions and stored procedures.

TABLE (SQL2003 Datatype: none)

A special datatype that stores a result set for a later process. It is used solely in procedural processing and *cannot* be used in a *CREATE TABLE* statement. This datatype alleviates the need for temporary tables in many applications. It can reduce the need for procedure recompiles, thus speeding execution of stored procedures and user-defined functions.

TEXT (SQL2003 Datatype: *CLOB*)

Stores very large passages of text up to 2,147,483,647 characters in length. *TEXT* and *IMAGE* values are often more difficult to manipulate than, say, *VARCHAR* values. For example, you cannot place an index on a *TEXT* or *IMAGE* column. *TEXT* can be manipulated using the functions *DATALength*, *PATINDEX*, *SUBSTRING*, *TEXTPTR*, and *TEXTVALID* as well as the commands *READTEXT*, *SET TEXTSIZE*, *UPDATETEXT*, and *WRITETEXT*.

TIMESTAMP (SQL2003 Datatype: *TIMESTAMP*)

Stores an automatically generated binary number that guarantees uniqueness in the current database and is therefore different from the ANSI *TIMESTAMP* datatype. *TIMESTAMPS* consume 8 bytes of storage space. *ROWVERSION* is now preferred over *TIMESTAMP* to uniquely track each row.

TINYINT (SQL2003 Datatype: none)

Stores unsigned integers between 0 and 255 in 1 byte of storage space. See the entry for *INT* to see rules about the *IDENTITY* property, which applies to this type.

UNIQUEIDENTIFIER (SQL2003 Datatype: none)

Represents a value that is globally unique across all databases and all servers. Values are represented as *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, in which each *x* is a hexadecimal digit in the range 0-9 or a-f. The only operations allowed against *UNIQUEIDENTIFIER*s are comparisons and NULL checks. Column constraints and properties are allowed on *UNIQUEIDENTIFIER* columns, with the exception of the *IDENTITY* property.

VARBINARY[(*n*)] (SQL2003 Datatype: *BLOB*)

Describes a variable-length binary value up to 8,000 bytes in size. Storage space consumed is equivalent to the size of the data inserted, plus 4 bytes.

VARCHAR[(*n*)], *CHAR VARYING*[(*n*)], *CHARACTER VARYING* [(*n*)] (SQL2003 Datatype: *CHARACTER VARYING*(*n*))

Holds fixed-length character data of 1 to 8,000 characters in length. Storage space is the actual size of the value entered in bytes, not the value of *n*.

2.4 Constraints

Constraints allow you to automatically enforce the integrity of data and to filter the data that is placed in a database. In a sense, *constraints* are rules that define which data values are valid during *INSERT*, *UPDATE*, and *DELETE* operations. When a data modification transaction breaks the rules of a constraint, the transaction is rejected.

In the ANSI standard, there are four constraint types: *CHECK*, *PRIMARY KEY*, *UNIQUE*, and *FOREIGN KEY*. (The RDBMS platforms may allow more. Refer to different platform sections of the [CREATE/ALTER TABLE Statement](#) to find out these exceptions.)

2.4.1 Scope

Constraints may be applied at the column-level or the table-level:

Column-level constraints

Declared as part of a column definition and apply only to that column.

Table-level constraints

Declared independently from any column definitions (traditionally, at the end of a *CREATE TABLE* statement) and may apply to one or more columns in the table. A table constraint is required when you wish to define a constraint that applies to more than one column.

2.4.2 Syntax

Constraints are defined when you create or alter a table. The general syntax for constraints is shown here:

```
CONSTRAINT [constraint_name] constraint_type [(column [...])]
[predicate] [constraint_deferment] [deferment_timing]
```

The syntax elements are as follows:

CONSTRAINT [*constraint_name*]

Begins a constraint definition and, optionally, provides a name for the constraint. When you omit *constraint_name*, the system will create one for you automatically. On some platforms, like DB2, you may omit the *CONSTRAINT* keyword as well. System-generated names are often incomprehensible. It is good practice to specify human-readable, sensible names for constraints.

constraint_type

Declares the constraint as one of the allowable types: *CHECK*, *PRIMARY KEY*, *UNIQUE*, or *FOREIGN KEY*. More information about each type of constraint appears later in this section.

column [...]

Associates one or more columns with the constraint. Specify the columns in a comma-delimited list. The column list must be enclosed in parentheses. The column list should be omitted for column-level constraints. Columns are not used in every constraint. For example, *CHECK* constraints do not generally use column references.

predicate

Defines a predicate for *CHECK* constraints.

constraint_deferment

Declares a constraint as *DEFERRABLE* or *NOT DEFERRABLE*. When a constraint is deferrable, you can specify that it be checked for a rules violation at the end of a transaction. When a constraint is not deferrable, it is checked for a rules violation at the conclusion of every SQL statement.

deferment_timing

Declares a deferrable constraint as *INITIALLY DEFERRED* or *INITIALLY IMMEDIATE*. When set to *INITIALLY DEFERRED*, the constraint check time will be deferred until the end of a transaction, even if it is composed of many SQL statements. In this case, the constraint must also be *DEFERRABLE*. When set to *INITIALLY IMMEDIATE*, the constraint check time is at the end of every SQL statement. In this case, the constraint may be either *DEFERRABLE* or *NOT DEFERRABLE*. The default is *INITIALLY IMMEDIATE*.

Note that the vendor platforms may have some variations on this syntax. Check different platform sections of [CREATE/ALTER TABLE Statement](#) for more details.

2.4.3 PRIMARY KEY Constraints

A *PRIMARY KEY* constraint declares one or more columns whose values uniquely identify each record in the table. It is considered a special case of the *UNIQUE* constraint. Some rules about primary keys:

- Only one primary key may exist on a table at a time.
- Columns in the primary key cannot contain datatypes of *BLOB*, *CLOB*, *NCLOB*, or *ARRAY*.
- Primary keys may be defined at the column level for a single column key or at the table level if multiple columns make up the primary key.
- Values in the primary key column(s) must be unique and not NULL.
- In a multicolumn primary key, called a *concatenated key*, the combination of values in all of the key columns must be unique and not NULL.
- Foreign keys can be declared that reference the primary key of a table to establish direct relationships between tables (or possibly, though rarely, within a single table).

The following ANSI standard code includes the options for creating both a table- and column-level primary key constraint on a table called **distributors**. The first example shows a column-level primary-key constraint, while the second shows a table-level constraint:

-- Creating a column-level constraint

```
CREATE TABLE distributors
```

```
  (dist_id   CHAR(4)  NOT NULL PRIMARY KEY,
```

```
   dist_name  VARCHAR(40),
```

```
   dist_address1 VARCHAR(40),
```

```
   dist_address2 VARCHAR(40),
```

```
   city       VARCHAR(20),
```

```
   state      CHAR(2)  ,
```

```
   zip        CHAR(5)  ,
```

```
   phone      CHAR(12) ,
```

```
   sales_rep  INT      );
```

-- Creating a table-level constraint

```
CREATE TABLE distributors
```

```
  (dist_id   CHAR(4)  NOT NULL,
```

```
   dist_name  VARCHAR(40),
```

```
dist_address1 VARCHAR(40),
dist_address2 VARCHAR(40),
city          VARCHAR(20),
state         CHAR(2)  ,
zip           CHAR(5)  ,
phone         CHAR(12) ,
sales_rep     INT      ,
```

CONSTRAINT pk_dist_id PRIMARY KEY (dist_id));

In the example showing a table-level primary key, we could easily have created a concatenated key by listing several columns separated by commas between each one.

2.4.4 FOREIGN KEY Constraints

A *FOREIGN KEY* constraint defines one or more columns in the table as referencing columns to a *UNIQUE* or *PRIMARY KEY* in another table. (A foreign key can reference a unique or primary key in the same table as the foreign key itself, but such foreign keys are rare.) Foreign keys can then prevent the entry of data into a table where there is no matching value in the related table. They are the primary means of identifying the relationship between tables in a relational database. Some rules about foreign keys:

- Many foreign keys may exist on a table at a time.
- A foreign key can be declared to reference either the primary key or a unique key of another table to establish a direct relationship between the two tables.

The full SQL2003 syntax for foreign keys is more elaborate than the general syntax for constraints shown earlier, and is dependent on whether you are making a table-level or column-level declaration:

-- Table-level foreign key

```
[CONSTRAINT [constraint_name] ]
FOREIGN KEY (local_column [,...])
REFERENCES referenced_table [ (referenced_column [,...]) ]
[MATCH {FULL | PARTIAL | SIMPLE} ]
[ON UPDATE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[ON DELETE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[constraint_deferment] [deferment_timing]
```

-- Column-level foreign key

```
[CONSTRAINT [constraint_name] ]
REFERENCES referenced_table [ (referenced_column [,...]) ]
```

[MATCH {FULL | PARTIAL | SIMPLE}]

[ON UPDATE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT}]

[ON DELETE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT}]

[*constraint_deferment*] [*deferment_timing*]

Keywords common to a standard constraint declaration are described above, under the "Syntax" section. Keywords specific to foreign keys are described in the following list:

FOREIGN KEY (*local_column* [,...])

Declares one or more columns of the table being created or altered and are subject to the foreign key constraint. This syntax is used *only* in table-level declarations and is excluded from column-level declarations. We recommend that the ordinal position and datatype of each column in the *local_column* list match comparably with the ordinal position and datatype of the columns in the **referenced_column** list.

REFERENCES *referenced_table* [(*referenced_column* [,...])]

Names the table and, where appropriate, the column(s) that hold the valid list of values for the foreign key. A **referenced_column** must already be named in a *NOT DEFERRABLE PRIMARY KEY* or *NOT DEFERRABLE UNIQUE KEY*. The table types must match. For example, if one is a local temporary table then both must be local temporary tables.

MATCH {FULL | PARTIAL | SIMPLE}

Defines the degree of matching required between the local and referenced columns in foreign key constraints when NULLs are present.

FULL

Declares that a match is acceptable when: 1) none of the referencing columns are NULL and match all of the values of the referenced column, or 2) all of the referencing columns are NULL. In general, you should use *MATCH FULL* or else ensure that all columns involved have NOT NULL constraints.

PARTIAL

Declares that a match is acceptable when at least one of the referenced column is NULL and the others match the corresponding referenced columns.

SIMPLE

Declares that a match is acceptable when any of the values of the referencing column are NULL or a match. This is the default.

ON UPDATE

Specifies that, when an *UPDATE* operation affects one or more **referenced_columns** of the primary or unique key on the referenced table, a corresponding action should be taken to ensure the foreign key does not lose data integrity. *ON UPDATE* may be declared independently of or with the *ON DELETE* clause. When omitted, the default is *ON UPDATE NO ACTION*.

ON DELETE

Specifies that when a *DELETE* operation affects one or more **referenced_columns** of the primary or unique key on the referenced table, that a corresponding action should be taken to ensure the foreign key does not lose data integrity. *ON DELETE* may be declared independent of or with the *ON UPDATE* clause. When omitted, the default for the ANSI standard is *ON DELETE NO ACTION*.

NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT

Defines the action the database takes to maintain data integrity of the foreign key when a referenced primary or unique key constraint value is changed or deleted.

NO ACTION

Tells the database to do nothing when a primary key or unique key value, referenced by a foreign key, is changed or deleted.

CASCADE

Tells the database to perform the same action (i.e., *DELETE* or *UPDATE*) on the matching foreign key when a primary key or unique key value is changed or deleted.

RESTRICT

Tells the database to prevent changes to the primary key or unique key value, referenced by the foreign key.

SET NULL

Tells the database to set the value in the foreign key to NULL when a primary key or unique key value is changed or deleted.

SET DEFAULT

Tells the database to set the value in the foreign key to the default (using default values you specify for each column) when a primary key or unique key value is changed or deleted.

As with the code example for primary keys, you can adapt this generic syntax to both column-level and table-level foreign key constraints. Note that column-level and table-level constraints perform their function in exactly the same way. They are merely defined at different levels of the *CREATE TABLE* command. In the following example, we create a single-column foreign key on the **salesrep** column referencing the **empid** column of the **employee** table. We create the foreign key two different ways, the first time at the column level, the second time at the table level:

-- Creating a column-level constraint

CREATE TABLE distributors

```
(dist_id    CHAR(4)  PRIMARY KEY,  
dist_name  VARCHAR(40),  
dist_address1 VARCHAR(40),  
dist_address2 VARCHAR(40),  
city       VARCHAR(20),  
state      CHAR(2)  ,  
zip        CHAR(5)  ,  
phone      CHAR(12) ,  
sales_rep  INT       NOT  
NULL REFERENCES employee(empid));
```

```
-- Creating a table-level constraint
CREATE TABLE distributors
  (dist_id    CHAR(4)  NOT NULL,
   dist_name  VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city       VARCHAR(20),
   state      CHAR(2)  ,
   zip        CHAR(5)  ,
   phone      CHAR(12) ,
   sales_rep  INT      ,
  CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
  CONSTRAINT fk_empid  FOREIGN KEY (sales_rep)
    REFERENCES employee(empid));
```

2.4.5 UNIQUE Constraints

A *UNIQUE* constraint, sometimes called a *candidate key*, declares that the values in one column, or the combination of values in more than one column, must be unique. Rules concerning unique constraints include:

- Columns in a unique key cannot contain datatypes of *BLOB*, *CLOB*, *NCLOB*, or *ARRAY*.
- The column or columns in a unique key may not be identical to those in any other unique keys, or to any columns in the primary key of the table.
- A single NULL value, if the unique key allows NULL values, is allowed.
- SQL2003 allows you to substitute the column list, shown in the general syntax diagram for constraints, with the keyword (*VALUE*). *UNIQUE (VALUE)* indicates that all columns in the table are part of the unique key. The *VALUE* keyword also disallows any other unique or primary keys on the table.

In the following example, we limit the number of distributors we do business with to only one distributor per ZIP Code. We also allow one (and only one) "catch-all" distributor with a NULL ZIP Code. This functionality can be implemented easily using a *UNIQUE* constraint, either at the column or the table level:

```
-- Creating a column-level constraint
CREATE TABLE distributors
  (dist_id    CHAR(4)  PRIMARY KEY,
   dist_name  VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city       VARCHAR(20),
   state      CHAR(2)  ,
   zip        CHAR(5)  UNIQUE ,
```

```
    phone    CHAR(12) ,
    sales_rep INT     NOT NULL
REFERENCES employee(empid));

-- Creating a table-level constraint
CREATE TABLE distributors
    (dist_id    CHAR(4) NOT NULL,
    dist_name  VARCHAR(40),
    dist_address1 VARCHAR(40),
    dist_address2 VARCHAR(40),
    city      VARCHAR(20),
    state     CHAR(2)  ,
    zip       CHAR(5)  ,
    phone     CHAR(12) ,
    sales_rep INT      ,
    CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
    CONSTRAINT fk_emp_id  FOREIGN KEY (sales_rep)
        REFERENCES employee(empid),
CONSTRAINT unq_zip    UNIQUE (zip) );
```

2.4.6 CHECK Constraints

CHECK constraints allow you to perform comparison operations to ensure that values match specific conditions that you set out. The syntax for a check constraint is very similar to the general syntax for constraints:

```
[CONSTRAINT] [constraint_name] CHECK (search_conditions)
[constraint_deferment] [deferment_timing]
```

Other elements of the constraint are introduced earlier in this section. The following element is unique to the *CHECK* constraint:

search_conditions

Specifies one or more search conditions that constrain the values inserted into the column or table, using one or more expressions and a predicate. Multiple search conditions may be applied to a column in a single check constraint using the *AND* and *OR* operators. (Think of a *WHERE* clause.)

A check constraint is considered matched when the search conditions evaluate to TRUE or UNKNOWN. Check constraints are limited to Boolean operations (e.g., =, >=, <=, or <>), though they may include any SQL2003 predicates such as *IN* or *LIKE*. Check constraints may be appended to one another (when checking a single column) using the *AND* and *OR* operators. Some other rules about *CHECK* constraints:

- A column or table may have one or more *CHECK* constraints.
- A search condition cannot contain aggregate functions, except in a subquery.
- A search condition cannot use non-deterministic functions or subqueries.

- A check constraint can only reference like objects. So if a check constraint is declared on a global temporary table, it cannot then reference a permanent table.
- A search condition cannot reference these ANSI functions: *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *CURRENT_TIMESTAMP*, *LOCALTIME*, and *LOCALTIMESTAMP*.

The following example adds a check constraint to the **dist_id** and **zip** columns. (This example uses generic code run on SQL Server.) The ZIP Code must fall into the normal ranges for postal ZIP Codes, while the **dist_id** values are allowed to contain either four alphabetic characters or two alphabetic and two numeric characters:

-- Creating column-level CHECK constraints

CREATE TABLE distributors

(dist_id CHAR(4)

CONSTRAINT pk_dist_id PRIMARY KEY

CONSTRAINT ck_dist_id CHECK

(dist_id LIKE '[A-Z][A-Z][A-Z][A-Z]' OR

dist_id LIKE '[A-Z][A-Z][0-9][0-9]'),

dist_name VARCHAR(40),

dist_address1 VARCHAR(40),

dist_address2 VARCHAR(40),

city VARCHAR(20),

state CHAR(2)

CONSTRAINT def_st DEFAULT ("CA"),

zip CHAR(5)

CONSTRAINT unq_dist_zip UNIQUE

CONSTRAINT ck_dist_zip CHECK

(zip LIKE '[0-9][0-9][0-9][0-9][0-9]'),

phone CHAR(12),

sales_rep INT

NOT NULL DEFAULT USER REFERENCES employee(emp_id))



Chapter 3. SQL Statement Command Reference

This chapter is the heart of *SQL in a Nutshell*, Second Edition. It is an alphabetical listing of SQL commands with detailed explanations and examples. Each command and function is identified in [Table 3-1](#) as being "supported," "supported with variations," "supported with limitations," or "not supported," for each of the five SQL dialects covered in this book: DB2, MySQL, Oracle, PostgreSQL, SQL Server. After a brief description of the SQL2003 standard, each vendor application is discussed briefly but thoroughly, with supporting examples. If a specific platform does not support a particular command, that fact will be noted in the table that heads the command and then the command will not be listed for that platform. Similarly, although *SQL in a Nutshell*, Second Edition, is not a comprehensive review of the SQL2003 standard, each command has been validated against SQL2003.

3.1 How to Use This Chapter

When researching a command in this chapter:

1. Read [Section 3.2](#).
2. Check the platform support table.
3. Read the section on SQL2003 syntax and description even if you are looking for a specific platform implementation.
4. Finally, read the specific platform implementation information.

Any common features between the platform implementations of a command are discussed and compared against the SQL2003 topic. Thus, the subsection on a platform's implementation of a particular command may not describe every aspect of that command, since some of its details may be covered in the SQL2003 section. Please note that if there is a keyword that appears in a command syntax but not in its keyword description, this is because we chose not to repeat descriptions that appear under the ANSI entry.



If you skip straight to a platform implementation and do not see a discussion of a keyword or clause that you are interested in, that means it is described (in common across all supporting platforms) under the SQL2003 entry of the command.

3.2 SQL Platform Support

[Table 3-1](#) provides a listing of the SQL statements, the platforms that support them, and the degree to which they support them. The following list offers useful tips for reading [Table 3-1](#), as well as what each abbreviation stands for. The sections that follow describe the commands from the table in detail:

1. The first column contains the alphabetized SQL commands.
2. The SQL statement class for each command is indicated in the second column.
3. The subsequent columns list the vendor's level of support:

Supported (S)

The platform supports the SQL2003 standard for the particular command.

Supported, with variations (SWV)

The platform supports the SQL2003 standard for the particular command, using vendor-specific code or syntax.

Supported, with limitations (SWL)

The platform supports some but not all of the functions specified by the SQL2003 standard for the particular command.

Not supported (NS)

The platform does not support the particular command according to the SQL2003 standard.

Remember that even if a specific SQL2003 command is listed as "Not supported," the platform usually has alternative coding or syntax to enact the same command or function. Therefore, be sure to read the discussion and examples for each command later in this chapter. Likewise, a few of the commands in [Table 3-1](#) are not found in the SQL2003 standard, and these have been indicated with the term "Non-ANSI" under the heading "SQL2003 class" in the table.

Conversely, a few ANSI commands in the table (such as *CREATE DOMAIN* and *ALTER DOMAIN*) are not currently supported by any of the database platforms covered in this text. Since this book focuses on the implementation of the SQL language, unsupported ANSI commands are shown in [Table 3-1](#) but are not documented elsewhere in the book.

Table 3-1. Alphabetical quick SQL command reference

SQL2003 Command	SQL2003 class	DB2 v8.1	MySQL v4.0	Oracle v10g	Postgre SQL v7.2	SQL Server 2000
<i>ALL/ANY/SOME</i>	SQL-data	S	NS	S	S	S
<i>BETWEEN</i>	SQL-data	S	S	S	S	S
<i>ALTER DATABASE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>ALTER DOMAIN</i>	SQL-schema	NS	NS	NS	NS	NS
<i>ALTER FUNCTION</i>	SQL-schema	SWV	SWL	SWV	SWL	SWV
<i>ALTER INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>ALTER METHOD</i>	SQL-schema	SWV	NS	NS	NS	NS
<i>ALTER PROCEDURE</i>	SQL-schema	SWV	SWL	SWV	SWL	SWV
<i>ALTER TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV

<i>ALTER TRIGGER</i>	Non-ANSI	SWV	NS T	SWV	SWV	SWV
<i>ALTER TYPE</i>	SQL-schema	SWV	NS	SWV	NS	NS
<i>ALTER VIEW</i>	Non-ANSI	SWV	NS	SWV	SWV	SWV
<i>CALL</i>	SQL-control	SWV	NS T	SWV	NS	NS
<i>CLOSE CURSOR</i>	SQL-data	SWV	NS	S	S	SWV
<i>COMMIT</i>	SQL-transaction	S	SWV	SWV	SWV	SWV
<i>CONNECT</i>	SQL-connection	SWV	NS	S	NS	SWV
<i>CREATE DATABASE</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>CREATE DOMAIN</i>	SQL-schema	NS	NS	NS	NS	NS
<i>CREATE FUNCTION</i>	SQL-schema	SWV	SWL	SWV	SWL	SWV
<i>CREATE INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>CREATE METHOD</i>	SQL-schema	SWV	NS	NS	NS	NS
<i>CREATE PROCEDURE</i>	SQL-schema	S	NS	S	NS	S
<i>CREATE ROLE</i>	SQL-schema	NS	NS	SWV	NS	NS
<i>CREATE SCHEMA</i>	SQL-schema	SWL	NS	SWV	NS	SWL
<i>CREATE TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>CREATE TRIGGER</i>	SQL-schema	SWV	NS	SWV	SWV	SWV
<i>CREATE TYPE</i>	SQL-schema	SWV	NS	SWL	SWV	NS
<i>CREATE VIEW</i>	SQL-schema	SWV	NS	SWV	SWV	SWV
<i>DECLARE CURSOR</i>	SQL-data	SWL	NS	SWL	SWL	SWL
<i>DELETE</i>	SQL-data	SWV	SWV	SWV	S	SWL
<i>DISCONNECT</i>	SQL-connection	SWL	NS	SWL	NS	SWL
<i>DROP DATABASE</i>	Non-ANSI	NS	SWV	S	SWV	SWV
<i>DROP DOMAIN</i>	SQL-schema	NS	NS	NS	NS	NS
<i>DROP FUNCTION</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>DROP INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>DROP METHOD</i>	SQL-schema	SWV	NS	SWV	SWV	NS
<i>DROP PROCEDURE</i>	SQL-schema	SWV	NS	S	NS	S
<i>DROP ROLE</i>	SQL-schema	NS	NS	SWV	NS	NS
<i>DROP TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV

	Schema					
<i>DROP TYPE</i>	SQL-schema	S	NS	S	S	NS
<i>DROP TRIGGER</i>	SQL-schema	SWV	NS	SWV	SWV	SWV
<i>DROP VIEW</i>	SQL-schema	S	NS	S	S	S
<i>EXCEPT</i>	SQL-data	SWL	NS	SWL	SWL	NS
<i>EXISTS</i>	SQL-data	S	S	S	S	S
<i>FETCH</i>	SQL-data	SWL	NS	SWL	SWV	SWV
<i>GRANT</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>IN</i>	SQL-data	S	S	SWV	S	S
<i>INSERT</i>	SQL-data	SWV	SWV	SWV	S	SWV
<i>INTERSECT</i>	SQL-data	SWL	NS	SWL	SWL	NS
<i>IS</i>	SQL-data	S	S	S	S	S
<i>JOIN subclause</i>	SQL-data	SWL	SWV	S	SWV	SWL
<i>LIKE</i>	SQL-data	S	SWV	S	SWV	SWV
<i>MERGE</i>	SQL-data	S	NS	S	NS	NS
<i>OPEN</i>	SQL-data	SWV	NS	S	NS	S
<i>OPERATORS</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>ORDER BY</i>	SQL-data	SWL	SWL	SWV	SWV	SWL
<i>RELEASE SAVEPOINT</i>	SQL-transaction	SWV	NS	NS	NS	NS
<i>RETURN</i>	SQL-control	SWV	NS	S	SWL	S
<i>REVOKE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>ROLLBACK</i>	SQL-transaction	S	SWL	SWV	SWL	SWV
<i>SAVEPOINT</i>	SQL-transaction	SWV	NS	S	NS	SWL
<i>SELECT</i>	SQL-data	SWV (ANSI joins supported)	SWV (ANSI joins partially supported)	SWV (ANSI joins supported)	SWV (ANSI joins partially supported)	SWV (ANSI joins supported)
<i>SET</i>	SQL-session	S	S	NS	S	S
<i>SET CATALOG</i>	SQL-session	NS	NS	NS	NS	NS
<i>SET COLLATION</i>	SQL-session	NS	NS	NS	NS	NS
<i>SET CONNECTION</i>	SQL-connection	SWL	NS	NS	NS	SWL
<i>SET CONSTRAINT</i>	SQL-connection	NS	NS	SWV	SWV	NS
<i>SET DESCRIPTOR</i>	SQL-session	NS	NS	NS	NS	NS

3.3 SQL Command Reference

ALL/ANY/SOME Operators

The *ALL* operator performs a Boolean test of a subquery for the existence of a value in all rows. The *ANY* operator, and its synonym *SOME*, performs a Boolean test of a subquery for the existence of a value in any of the rows tested.

Platform	Command
DB2	Supported
MySQL	Not supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

SELECT ...

WHERE *expression comparison* {ALL | ANY | SOME} (*subquery*)

Keywords

WHERE *expression*

Tests a scalar expression, such as a column, against every value in the *subquery*, for *ALL*, and against every value until a match is found for *ANY* and *SOME*. All rows must match the expression to return a Boolean TRUE value for the *ALL* operator, while one or more rows must match the expression, for *ANY* and *SOME*, to return a Boolean TRUE value.

comparison

Compares the *expression* to the *subquery*. The *comparison* must be a standard comparison operator like =, <>, !=, >, >=, <, or <=.

Rules at a Glance

The *ALL* operator returns a Boolean TRUE value when one of two things happen: either the subquery returns an empty set (i.e., no records) or every record in the set meets the comparison. *ALL* returns FALSE when any records in the set do not match the value comparison. *ANY* and *SOME* operators return a Boolean TRUE when at least one record in the subquery matches the comparison operation, and FALSE when no record matches the comparison operation (or when a subquery returns an empty result set). If even one return value of the subquery is NULL, then the operation evaluates as NULL, not as TRUE.



Do not include special clauses like *ORDER BY*, *GROUP BY*, *CUBE*, *ROLLUP*, *WITH*, etc. in your subquery. Refer to [SUBQUERY Substatement](#) for more information.

For example, we want to see which authors currently have no titles:

```
SELECT au_id
FROM authors
WHERE au_id <> ALL(SELECT au_id FROM titleauthor)
```

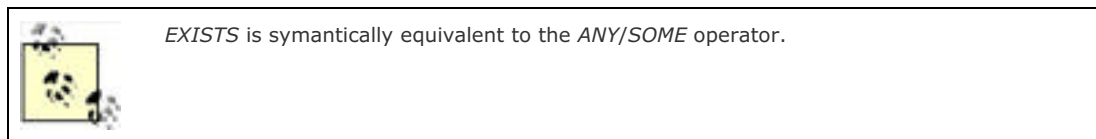
You can use *ANY* / *SOME* to perform filtering checks of different kinds. For example, the following query will retrieve any records from the **employees** table that exist in the **employees_archive** table that were located in Anchorage:

```
SELECT *
FROM employees
WHERE job_lvl = ANY(SELECT job_lvl FROM employees_archive
                    WHERE city = 'Anchorage')
```

The above query returns information about employees who have the same **job_lvl** as any employee working in Anchorage, Alaska.

Programming Tips and Gotchas

The *ALL* and *ANY/SOME* operators are somewhat difficult to get used to. Most developers find it a lot easier to use similar functions like *IN* and *EXISTS*.



Platform Differences

All of the platforms support *ALL* and *ANY/SOME* in the manner described above except MySQL, which doesn't support subqueries until Version 4.0. On pre-Version 4.0 MySQL platforms, consider using the *IN* operator instead of *EXISTS* for *ANY/SOME*.

Oracle supports one minor variation in that you can supply a list of values instead of a subquery. For example, we can find all employees who have a **job_lvl** value equal to 9 or 14:

```
SELECT * FROM employee
WHERE job_lvl = ALL(9, 14);
```

DB2 and SQL Server support some additional comparison operators: not greater than (!>) and not less than (!<).

See Also

- ALL/ANY/SOME
- BETWEEN
- EXISTS
- SELECT
- WHERE

BETWEEN Operator

The *BETWEEN* operator performs a Boolean test of a value against a range of values. It returns TRUE when the value is included in the range and FALSE when the value falls outside of the range. The results are NULL (unknown) if any of the range values are NULL.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

SELECT ...

WHERE *expression* [NOT] BETWEEN *lower_range* AND *upper_range*

Keywords

WHERE *expression*

Compares a scalar expression, such as a column, to the range of values bounded by *upper_range* and *lower_range*.

[NOT] BETWEEN *lower_range* AND *upper_range*

Compares the *expression* to the *lower_range* and *upper_range*. The comparison is inclusive, meaning that it is equivalent to saying "where *expression* is [not] greater than or equal to *lower_range* and less than or equal to *upper_range*."

Rules at a Glance

The *BETWEEN* operator is used to test an expression against a range of values. The *BETWEEN* operator may be used with any datatype except *BLOB*, *CLOB*, *NCLOB*, *REF*, and *ARRAY*.

For example, we want to see **title_ids** that have year-to-date sales between 10,000 and 20,000:

```
SELECT title_id
FROM titles
WHERE ytd_sales BETWEEN 10000 AND 20000
```

BETWEEN is inclusive of the range of values listed. In this case, it includes the values of 10,000 and 20,000 in the search. If you want an exclusive search, then you must use the greater than (>) and less than (<) symbols:

```
SELECT title_id
FROM titles
WHERE ytd_sales > 10000
AND ytd_sales < 20000
```

The *NOT* operator allows you to search for values outside of the *BETWEEN* range. For example, you can also find all the **title_ids** that were not published during 2002:

```
SELECT title_id
FROM titles
WHERE pub_date NOT BETWEEN '01-JAN-2003' AND '31-DEC-2003'
```

Programming Tips and Gotchas

Some coders are very particular about how the keyword *AND* is used in *WHERE* clauses. To prevent a casual reviewer from thinking that the *AND* used in a *BETWEEN* operation is a logical *AND* operator, you might want to use parentheses to encapsulate the entire *BETWEEN* clause:

```
SELECT title_id
FROM titles
WHERE (ytd_sales BETWEEN 10000 AND 20000)
AND pubdate >= '1991-06-12 00:00:00.000'
```

Platform Differences

All of the platforms support *BETWEEN* in the manner described above.

See Also

- ALL/ANY/SOME
- EXISTS
- SELECT
- WHERE

CALL Statement

The *CALL* statement invokes a stored procedure.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
CALL procedure_name ([parameter [...] ])
```

Keywords

```
CALL procedure_name
```

Specifies the stored procedure that you want to invoke. It must be a previously defined stored procedure that is available in the current user context (instance, database, schema, etc.).

(*parameter* [,...])

Provides values for each input parameter required by the stored procedure. Each parameter listed is required by the stored procedure in the same ordinal position. Thus, the fifth parameter listed would provide the value for the fifth argument required by the stored procedure. The parameters must be enclosed in parentheses and separated by commas. Note that the parentheses are required even if there are no parameter values (e.g., even if there are no parameter values, you must still use *CALL*()). Strings should be in single quotes. If the stored procedure has only *OUT* parameters, place host variables or parameter markers within the parentheses.

Rules at a Glance

The *CALL* statement makes it easy to invoke a stored procedure. Simply provide the name of the stored procedure and include any parameters used by the stored procedure, enclosing them within parentheses.

This Oracle example creates a simple stored procedure, and then calls it:

```
CREATE PROCEDURE update_employee_salary
```

```
(emp_id NUMBER, updated_salary NUMBER)
```

```
IS
```

```
BEGIN
```

```
    UPDATE employee SET salary = updated_salary
```

```
WHERE employee_id =emp_id ;
```

```
END;
```

```
CALL update_employee_salary(1517, 95000);
```

Programming Tips and Gotchas

The return status of a called stored procedure can be found, typically, by using *GET DIAGNOSTIC*. *GET DIAGNOSTIC* is not widely supported among the various database platforms, so check the platform documentation for more details.

Many platforms also support an alternative command called *EXECUTE* to perform the same functionality. In some cases, you may prefer *EXECUTE* to *CALL*, since the former can be used to execute any kind of prepared SQL, including methods, functions, or batches of SQL code.

DB2

DB2 supports *CALL* along with *DESCRIPTOR* capabilities:

```
CALL procedure_name { [(parameter [,...]) ] | USING DESCRIPTOR  
    descriptor_name }
```

where:

USING DESCRIPTOR *descriptor_name*

Identifies a SQL Descriptor Area (SQLDA) structure containing a valid description of the host variables. An SQLDA is a collection of variables that is required for execution of the SQL *DESCRIBE* statement. The SQLDA variables are options that can be used by the *PREPARE*, *OPEN*, *FETCH*, *EXECUTE*, and *CALL* statements. Values in the SQLDA must be set prior to issuing the *CALL* statement. This is a deprecated feature in DB2. We recommend that you instead use the standard *CALL proc(param, param)* syntax.

DB2 allows you to substitute host variables for the procedure name or the parameters, where appropriate. When using the *USING DESCRIPTOR* clause, the variables must be string variables of 254 or fewer characters.

On DB2, any return status issued by the called stored procedure is returned to the SQLERRD field of the SQLCA (SQL Communications Area). You can also use the *GET DIAGNOSTIC* statement to retrieve the *RETURN_STATUS* value (-1 indicates error) or row count.

MySQL

Not supported.

Oracle

Oracle allows the *CALL* statement to invoke standalone stored procedures, functions, and methods, as well as stored procedures and functions contained within a type or package. Following is the Oracle syntax:

```
CALL [schema][{type_name | package_name}.]procedure_name@dblink  
  
[(parameter [...] )]  
  
[INTO :variable_name [[INDICATOR] :indicator_name ]
```

where:

```
CALL [schema][{type_name | package_name}.] procedure_name@dblink
```

Calls the named object. You may fully enumerate the object name, including schema, type, etc., or allow Oracle to assume the current schema and database instance. If the procedure or function resides in another database, simply specify the database via a database link name, shown as *dblink* in the syntax. *dblink* must refer to a previously created database link.

```
INTO :variable_name
```

Specifies the name of a previously declared variable that will store the value returned when you call a function. *INTO* is required, and only usable, when calling a function.

```
INDICATOR :indicator_name
```

Retains the condition of the host variable, for example whether the return value is NULL, for functions precompiled in a Pro*C/C++ routine.

The parameters used in an Oracle *CALL* statement *may not* include pseudocolumns or the *VALUE* or *REF* functions. You must use a host variable for any parameter that corresponds to an *OUT* or *IN OUT* argument of the called stored procedure.

PostgreSQL

Not Supported.

SQL Server

Not supported. Instead, use the non-ANSI *EXECUTE* statement.

See Also

CREATE/ALTER PROCEDURE

CLOSE CURSOR Statement

The *CLOSE CURSOR* statement closes a server-side cursor created with a *DECLARE CURSOR* statement.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported

PostgreSQL	Supported
SQL Server	Supported, with variations

SQL2003 Syntax

CLOSE cursor_name

Keywords

cursor_name

Names a cursor previously created with the *DECLARE CURSOR* statement.

Rules at a Glance

The *CLOSE* statement closes a cursor and destroys the cursor result set. All the database platforms release any locks that were held by the cursor, though this is not specified in the ANSI standard. For example:

CLOSE author_names_cursor;

Programming Tips and Gotchas

You can also close a cursor implicitly using a *COMMIT* statement or, for cursors defined *WITH HOLD*, using a *ROLLBACK* statement.

DB2

DB2 supports an added optional *WITH* clause on the *CLOSE* statement:

CLOSE cursor_name [WITH RELEASE]

where:

WITH RELEASE

Attempts to release all read locks held by the cursor. Note that DB2 is not always able to release all locks since some locks may be in use by other operations.

The *WITH RELEASE* clause is ignored on cursors run under isolation level 3 CS or UR, when the cursor is defined in a method, when the cursor is defined in a function, or when the cursor is defined in a stored procedure that is called by a function or method. *WITH RELEASE* may allow phantom read and nonrepeatable read anomalies when run under isolation levels RS or RR. For a detailed explanation of isolation levels, including phantom reads and nonrepeatable reads, refer to *SET TRANSACTION ISOLATION LEVEL*.

PostgreSQL

PostgreSQL supports the ANSI standard syntax. PostgreSQL issues an implicit *CLOSE* statement for every open cursor when a transaction is ended with a *COMMIT* or *ROLLBACK* statement.

SQL Server

Microsoft SQL Server supports the ANSI standard syntax, and an additional *GLOBAL* keyword:

CLOSE [GLOBAL] cursor_name

where:

GLOBAL

Identifies the previously defined cursor as a global cursor.

In the ANSI standard behavior, closing a cursor destroys the cursor result set. Locking is a physical feature of each database platform and thus not a part of the ANSI SQL definition. However, all the database platforms covered here drop the locks taken up by the cursor. Another physical implementation detail is that SQL Server does not automatically reallocate memory structures consumed by a cursor to the memory pool. To accomplish such reallocation, you must issue a *DEALLOCATE cursor_name* command.

This example from Microsoft SQL Server opens a cursor and fetches a result set of all employees who have a last name starting with "K":

```
DECLARE employee_cursor CURSOR FOR
```

```
    SELECT lname, fname
```

```
    FROM pubs.dbo.employee
```

```
    WHERE lname LIKE 'K%'
```

```
OPEN employee_cursor
```

```
FETCH NEXT FROM employee_cursor
```

```
WHILE @@FETCH_STATUS = 0
```

```
BEGIN
```

```
    FETCH NEXT FROM employee_cursor
```

```
END
```

```
CLOSE employee_cursor
```

```
DEALLOCATE employee_cursor
```

```
GO
```

See Also

DECLARE CURSOR
FETCH
OPEN

COMMIT Statement

The *COMMIT* statement explicitly ends an open transaction and makes the changes permanent in the database. Transactions can be opened implicitly as part of an *INSERT*, *UPDATE*, or *DELETE* statement, or opened explicitly with a *START* statement. In either case, an explicitly issued *COMMIT* statement will end the open transaction.

Platform	Command
DB2	Supported
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

COMMIT [WORK] [AND [NO] CHAIN]

Keywords

COMMIT [WORK]

Ends the current, open transaction and writes any data manipulated by the transaction to the database. The optional keyword *WORK* is noise and has no effect.

AND [NO] CHAIN

AND CHAIN tells the DBMS to treat the next transaction as if it were a part of the preceding transaction. In effect, the two transactions are separate units of work, but they share a common transaction environment (such as transaction isolation level). Including the optional *NO* keyword tells the DBMS to explicitly use the ANSI default behavior. The *COMMIT* keyword by itself is functionally equivalent to the statement *COMMIT WORK AND NO CHAIN*.

Rules at a Glance

For simple operations, you will execute transactions (that is, SQL code that manipulates or changes data and objects in a database) without explicitly declaring a transaction. However, all transactions are best managed by explicitly closing them with a *COMMIT* statement. Because records and even entire tables can be locked for the duration of a transaction, it is extremely important that transactions are completed as quickly as possible. Thus, manually issuing a *COMMIT* statement with a transaction can help control user concurrency issues and locking problems on the database.

Programming Tips and Gotchas

The most important gotcha to consider is that some database platforms perform automatic and *implicit* transactions, while others require *explicit* transactions. If you assume a platform uses one method of transactions over the other, you may get bitten. Thus, when moving between database platforms you should follow a standard, preset way of addressing transactions. We recommend always using explicit transactions with *START TRAN*, on database platforms that support it, to begin a transaction, and *COMMIT* or *ROLLBACK* to end a transaction.

In addition to finalizing a single or group of data-manipulation operation(s), *COMMIT* has some interesting effects on other aspects of a transaction. First, it closes any associated open cursors. Second, any temporary table(s) specified with *ON COMMIT DELETE ROWS* (an optional clause of the *CREATE TABLE* statement) are cleared of data. Third, all deferred constraints are checked. If any deferred constraints are violated, the transaction is rolled back. Finally, all locks opened by the transaction are released. Please note that SQL2003 dictates that transactions are *implicitly opened* when one of the following statements is executed:

- ALTER
- CLOSE
- COMMIT AND CHAIN
- CREATE
- DELETE
- DROP
- FETCH
- FREE LOCATOR
- GRANT

- *HOLD LOCATOR*
- *INSERT*
- *OPEN*
- *RETURN*
- *REVOKE*
- *ROLLBACK AND CHAIN*
- *SELECT*
- *START TRANSACTION*
- *UPDATE*

If you did not explicitly open a transaction when you started one of the commands listed above, the standard dictates that the DBMS platform open a transaction for you.

DB2

DB2 supports the standard, but not the *AND [NO] CHAIN* clause.

MySQL

MySQL supports *COMMIT* and transactions only with InnoDB tables. It does not support the *AND [NO] CHAIN* clause.

Oracle

Oracle supports the standard, but not the *AND [NO] CHAIN* clause. In addition, Oracle has a couple of extensions to the standard clause:

```
COMMIT [WORK] [ {COMMENT 'text' | FORCE 'text' [, int]} ];
```

where:

COMMENT '*text*'

Associates a comment with the current transaction where '*text*' is a literal string up to 255 characters long. The text string is stored in the Oracle data dictionary view **DBA_2PC_PENDING** with the transaction ID, in case the transaction rolls back.

FORCE '*text*' [, *int*]

Allows an in-doubt distributed transaction to be manually committed, where '*text*' is a literal string that identifies the local or global transaction ID. Transactions can be identified by querying the Oracle data dictionary view **DBA_2PC_PENDING**. The optional *int* parameter is an integer that explicitly assigns a system change number (SCN) to the transaction. Without *int*, the transaction commits using the current SCN.

Issuing a *COMMIT* statement with the *FORCE* clause will commit only the transaction explicitly identified in the *FORCE* clause. It will not affect the current transaction unless it is explicitly identified. The *FORCE* clause is not usable in PL/SQL statements.

The following example commits a transaction while associating a comment with the transaction:

```
COMMIT WORK COMMENT 'In-doubt transaction, Call (949) 555-1234';
```

PostgreSQL

PostgreSQL implements the following syntax:

```
COMMIT [WORK | TRANSACTION];
```

In PostgreSQL, both the *WORK* and *TRANSACTION* keywords are optional. When you issue a *COMMIT*, all open transactions are written to disk and the results of those transactions then become visible to other users. For example:

```
INSERT INTO sales VALUES('7896','JR3435','Oct 28 1997',25,  
    'Net 60','BU7832');  
  
COMMIT WORK;
```

SQL Server

SQL Server does support the *AND [NO] CHAIN* clause. SQL Server supports the keyword *TRANSACTION* as an equivalent to *WORK*, as in the following syntax:

```
COMMIT { [TRAN[SACTION] [transaction_name] ] | [WORK] }
```

Microsoft SQL Server allows the creation of a specific, named transaction using the *START TRAN* statement. The *COMMIT TRANSACTION* syntax allows you to specify an explicit, named transaction to close or to store a transaction name in a variable. Curiously, SQL Server still commits only the most recently opened transaction, despite the name of the transaction that is specified.

When you issue *COMMIT WORK*, SQL Server terminates all open transactions and writes their changes to the database. You *may not* specify a transaction name when using *COMMIT WORK*.

See Also

ROLLBACK
START TRANSACTION

CONNECT

The *CONNECT* statement establishes a connection to the DBMS and to a specific database within the DBMS.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported, with variations

SQL2003 Syntax

```
CONNECT TO {DEFAULT | {[server_name] [AS connection_name]}  
    [USER user_name] }
```

Keywords

DEFAULT

Initiates a default session with the server where the user authorization is the default and the current database is the default. The standard specifies that *CONNECT TO DEFAULT* is issued implicitly if you start a SQL session without first issuing a *CONNECT* statement.

server_name

Establishes a connection to a named server. The *server_name* may be a string literal enclosed in single-quotes or a host variable.

AS *connection_name*

Establishes a connection with the name *connection_name* shown in single-quotes. The optional *connection_name* may be a string literal enclosed in single quotes or a host variable. It is optional only for the first connection attached to a server. All subsequent sessions must include the *AS* clause. This is useful for telling user connections apart when many users, perhaps even a single user, have many open sessions on a given *server_name*.

USER *user_name*

Establishes a connection to the named server under the *user_name* specified.

Rules at a Glance

Issue the *CONNECT* statement to open an interactive SQL session with a DBMS. The period between issuing the *CONNECT* and *DISCONNECT* statements is commonly called a *session*. Typically, you will complete all work on a DBMS during an explicitly invoked session.

If you do not specify the server name, the connection name, or the username, then the platform DBMS will provide default values for you. The defaults vary from platform to platform.

To connect to the "houston" server under a specific user ID, you might issue the command:

```
CONNECT TO houston USER pubs_admin
```

If the DBMS requires named connections, you might use the following, alternative syntax:

```
CONNECT TO houston USER pubs_admin AS  
pubs_administrative_session;
```

If you want just a simple, short-term connection then you might use the default:

```
CONNECT TO DEFAULT;
```

Programming Tips and Gotchas

If the *CONNECT* statement is invoked without explicitly disconnecting, the old session becomes dormant and the new session becomes active. You can then switch between connections using the *SET CONNECTION* statement.



The Oracle tool SQL*Plus uses the *CONNECT* command somewhat differently: to connect a user to a specific schema.

DB2 supports a couple of variations to the ANSI standard:

```
CONNECT { [TO] [{server_name] [IN SHARE MODE |  
    IN EXCLUSIVE MODE [ON SINGLE DBPARTITIONNUM] | [RESET] ] }  
[USER user_name ] { [USING password] | [NEW password  
    CONFIRM password]} }
```

where:

CONNECT

Equates to the SQL statement *CONNECT TO DEFAULT*, when used alone.

TO server_name

Identifies the server to which the connection will be established. The *server_name* may be a string literal or a character datatype variable of 8 or fewer characters (UDB) or 16 or fewer characters (OS/390 or z/OS).

IN SHARE MODE

Establishes a concurrent connection and allows other users to connect to the database. Prevents other users from connecting in exclusive mode.

IN EXCLUSIVE MODE [ON SINGLE DBPARTITIONNUM]

Acquires an exclusive connection on the application server and prevents any new users from connecting unless they use the same *user_name* and *password*. The optional *ON SINGLE DBPARTITIONNUM* clause is only usable on a partitioned database. This option specifies that the coordinator database partition is connected in exclusive mode, while all other database partitions are connected in share mode.

RESET

Executes a *COMMIT* statement, then disconnects the session from the server.

USER user_name

Establishes the connection under the named user account. The *user_name* may be a string literal or a character datatype variable of 8 or fewer characters in length.

USING password

Specifies the *password* for the user account of *user_name*. It may be a string literal or a character datatype variable of 8 or fewer characters in length.

NEW password CONFIRM password

Specifies a new *password* that will be assigned to the user account called *user_name*. It may be a string literal or a character datatype variable of 18 or fewer characters in length.

For example, we might want to change the password for the user account *sam* to *mephrun* using a *VARCHAR(8)* host variable, *app_server*, to store the value of the application server:

```
CONNECT TO :app_server USER sam NEW 'mephrun' CONFIRM 'mephrun';
```

MySQL

Not supported.

Oracle

The *CONNECT* clause allows a database connection as a specific username. Alternately, a connection can be established for special privileges with *AS SYSOPER* or *AS SYSDBA*. The Oracle syntax is:

```
CONN[ECT] [[username/password] {AS {SYSOPER | SYSDBA} ] ]
```

where:

```
CONN[ECT] username/password
```

Establishes a connection to the database instance.

```
AS { SYSOPER | SYSDBA }
```

Establishes the connection as one of the two optional system roles.

If another connection is already open, *CONNECT* commits any open transactions, closes the current session, and opens the new one.

Oracle also allows the *CONNECT* statement in its SQL*Plus and iSQL*Plus tools.

PostgreSQL

PostgreSQL does not explicitly support the *CONNECT* statement. However, it does support the statement *SPI_CONNECT* under the Server Programming Interface and *PG_CONNECT* under the PG/TCL programming package.

SQL Server

SQL Server supports the basic elements of the *CONNECT* statement within Embedded SQL (inside of C++ or Visual Basic programs), as in the following:

```
CONNECT TO [server_name.]database_name [AS connection_name]
```

```
USER {login_name[.password] | $integrated}
```

where:

```
CONNECT TO server_name.database_name
```

Specifies the server and database to which you want a connection. You may leave off the *server_name* if you wish to default to the local server.

```
AS connection_name
```

Names the connection as an alphanumeric string up to 30 characters in length. Symbols are allowed, except for hyphens (-), but the first character must be a letter. *CURRENT* and *ALL* are reserved and may not be used for a *connection_name*. You need only define connection names when making more than one connection.

```
USER {login_name[.password] | $integrated}}
```

Connects the session under the *login_name* specified using the *password* provided or using Windows integrated security. The password is optional when supplying a login name.

For example, we can connect to the server named *new_york* as the Windows login *pubs_admin*:

```
CONNECT TO new_york.pubs USER pubs_admin
```

To issue the same command under SQL Server standard security:

```
EXEC SQL CONNECT TO new_york.pubs USER pubs_admin
```

To issue the same command under Windows integrated security:

```
EXEC SQL CONNECT TO new_york.pubs USER $integrated
```

To switch to a different connection, you should use the *SET CONNECTION* statement.

See Also

SET CONNECTION

CREATE/ALTER DATABASE Statement

The ANSI standard does not actually contain a *CREATE DATABASE* statement. However, since it is nearly impossible to operate a SQL database without this command, we've added *CREATE DATABASE* here. Almost all database platforms support some version of this command.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

Rules at a Glance

This command creates a new, blank database with a specific name. Most DBMS platforms require the user to possess administrator privileges in order to create a new database. Once the new database is created, you can populate it with database objects (such as tables, views, triggers, and so on) and populate the tables with data.

Depending on the platform, *CREATE DATABASE* may also create corresponding files on the filesystem that contain the data and metadata of the database.

Programming Tips and Gotchas

Since *CREATE DATABASE* is not an ANSI command, it is prone to rather extreme variation in syntax between platforms.

DB2

The DB2 statement *CREATE DATABASE* initializes a new database with a variety of user-defined characteristics, such as collation and autoconfiguration options. In addition, the statement also creates the three initial tablespaces, system tables, and recovery logs required by a DB2 database:

```
CREATE {DATABASE | DB} database_name
```

```
[AT DBPARTITIONNUM] [ON path_and_drive] [ALIAS database_alias]
```

```
[USING CODESET codeset TERRITORY territory]
```

```
[COLLATE USING {COMPATIBILTiy | IDENTITY | NLSCHAR | SYSTEM} ]
```

```
[NUMSEGS int] [DFT_EXTENT_SZ int]
```

```
[CATALOG TABLESPACE tablespace_definition]
```


[USER TABLESPACE *tablespace_definition*]

[TEMPORARY TABLESPACE *tablespace_definition*] [WITH "comments"]

[AUTOCONFIGURE [USING *param value* [...]]

[APPLY {DB ONLY | DB AND DBM | NONE}]

where:

CREATE {DATABASE | DB} *database_name*

Creates a database called *database_name*. The name must be unique from all other database names on either the local database directory or the system database directory.

AT DBPARTITIONNUM

Creates the database on the database partition where the command was issued. This command is used only to recreate a database that was dropped; it should not be used to create a brand new database. Once the *CREATE DATABASE AT DBPARTITIONNUM* statement is issued, the database must be immediately restored to the node.

ON *path_and_drive*

Specifies the path and drive, up to 205 characters long, to create the database on a Windows or Unix-based system. When this clause is omitted, the database is created on the default database path declared in the database manager configuration file *dftdbpath* parameter.

ALIAS *database_alias*

Declares an alias for the database within the system database directory.

USING CODESET *codeset* TERRITORY *territory*

Declares the *codeset* and *territory* used by the database. Once these values are set, they cannot be changed later.

COLLATE USING {COMPATIBLITY | IDENTITY | NLSCHAR | SYSTEM}

Specifies the type of collation used for the database. Once set, the collation sequence cannot be changed later. Options are:

COMPATIBILITY

Uses DB2 v2 collation sequence for backward compatibility.

IDENTITY

Uses the identity collation sequence, where strings are compared byte for byte.

NLSCHAR

Uses the collation sequence unique for the database codeset and territory. This option is currently only supported with the Thai code page (CP874).

SYSTEM

Uses the collation sequence for the current territory.

NUMSEGS *int*

Specifies the number of segment directories, as *int*, created to store DAT, IDX, LF, LB, and LBA files for any default System Managed Space (SMS) tablespaces. This setting does not affect DMS tablespaces, SMS tablespaces with explicitly defined creation characteristics, or SMS tablespaces created after the database is created.

DFT_EXTENT_SZ *int*

Specifies the default extent size, as *int*, of the tablespaces in the database.

CATALOG TABLESPACE *tablespace_definition*

Specifies the definition of the tablespace that holds the catalog tables. Details on the syntax for *tablespace_definition* are detailed below. When this clause is omitted, the catalog tables will be created in an SMS tablespace called **SYSCATSPACE** with the values set by the *NUMSEGS* and *DFT_EXTENT_SZ* clauses.

USER TABLESPACE *tablespace_definition*

Specifies the definition of the tablespace that holds the initial user tablespace. Details on the syntax for *tablespace_definition* are detailed below. When this clause is omitted, the user tablespace will be created in an SMS tablespace called **USERSPACE1** with the values set by the *NUMSEGS* and *DFT_EXTENT_SZ* clauses.

TEMPORARY TABLESPACE *tablespace_definition*

Specifies the definition of the tablespace that holds the initial system temporary tablespace. Details on the syntax for *tablespace_definition* are detailed below. When this clause is omitted, the user tablespace will be created in an SMS tablespace called **TEMPSPACE1** with the values set by the *NUMSEGS* and *DFT_EXTENT_SZ* clauses.

WITH "*comments*"

Specifies a description, up to 30 characters in length, for the database within the database directory.

AUTOCONFIGURE [USING *param value* [...]]

Declares user-defined settings for important database settings such as the buffer pool size. When issued without the *USING* clause, DB2 calculates the optimal settings and applies them. The list of *param* settings and their possible *values* are:

mem_percent

Defines the percentage of memory dedicated to the database. The default is 25, but may range from 1-100. Do not set this to 100 if other applications are running on the server.

workload_type

Defines the database workload as either '*simple*', '*mixed*', or '*complex*', with '*mixed*' as the default. The '*simple*' workload tunes I/O for transactions, while '*complex*' tunes the workload for more CPU intensive queries. '*Mixed*' provides tuning that is effective, though not optimized, for both workloads.

num_stmts

Defines the number of statements per unit of work. The default is 25, but may range from 1-1,000,000.

tpm

Defines the expect number of transactions per minute. The default is 60, but may range from 1-50,000.

admin_priority

Defines whether the database should be tuned for better performance (*value* = '*performance*') or better

recovery time (*value* = 'recovery'), or the middle ground (*value* = 'both'), which is the default.

num_local_apps

Defines the number of connected local applications. The default is 0, but may range from 0-5,000.

num_remote_apps

Defines the number of connections by remote applications. The default is 100, but may range from 0-5,000.

isolation

Defines the transaction isolation level of the applications connecting to the database. The default is RR, but valid values include RR, RS, CS, and UR. Refer to the entry on *SET TRANSACTION ISOLATION LEVEL* for more details on DB2's isolation levels.

bp_resizeable

Defines whether buffer pools may be resized or not. The default is yes, but it may also be set to no.

APPLY {DB ONLY | DB AND DBM | NONE}

Determines the scope of any autoconfiguration parameters set using the *AUTOCONFIGURATION* clause:

DB ONLY

Applies changes only to the database configuration and buffer pool settings.

DB AND DBM

Applies settings to the database manager configuration, the database configuration, and the buffer pool settings.

NONE

Displays the recommended settings, but does not apply them.

The *tablespace_definition* used by DB2 has its own syntax as well:

MANAGED BY

```
{SYSTEM USING ('container' [,...]) |  
DATABASE USING ({FILE | DEVICE} 'container' int [,...]) }  
[EXTENTSIZE int] [PREFETCHSIZE int] [OVERHEAD int]  
[TRANSFERRATE int]
```

where:

MANAGED BY

Defines whether the tablespace is managed by a system managed space (SMS) or a database managed space (DMS):

SYSTEM USING ('container' [,...])

Defines one or more *containers* for the SMS of up to 240 bytes in length. The *container* may be an absolute or relative directory name.

DATABASE USING ({FILE | DEVICE} '*container*' *int* [,...])

Defines one or more *containers* for the DMS of up to 254 bytes in length. The *container* may be an absolute or relative directory name. The type of container (*FILE* for a filename or *DEVICE* for a pre-existing device) and its size (*int*), plus a suffix of K (kilobytes), M (megabytes), or G (gigabytes) are also specified. If the file does not exist when specified, DB2 will create one. The device must already exist to be used.

EXTENTSIZE *int*

Defines the number of pages, *int*, that are written to the container before skipping to the next container. (The size of the pages is determined by the *PAGESIZE* setting.) Alternately, the extent size (*int*) may be declared in bytes by attaching a suffix of K (kilobytes), M (megabytes), or G (gigabytes) to the integer value.

PREFETCHSIZE *int*

Defines the number of pages, *int*, that are read from the tablespace when prefetching data. (The size of the pages are determined by the *PAGESIZE* setting.) Alternately, the extent size (*int*) may be declared in bytes by attaching a suffix of K (kilobytes), M (megabytes), or G (gigabytes) to the integer value.

OVERHEAD *int*

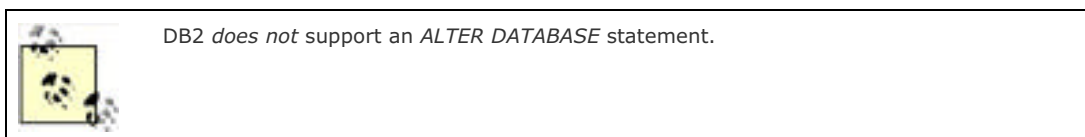
Defines the I/O controller overhead, disk seek, and latency time, in milliseconds, *int*, used to determine the cost of I/O during query optimization. The value for *int* should be an average for all containers that belong to the tablespace.

TRANSFERRATE *int*

Defines the time to read one page into memory, in milliseconds, *int*, used to determine the cost of I/O during query optimization. The value for *int* should be an average for all containers that belong to the tablespace.

Note that DB2 tablespaces may also be created using the DB2-specific command *CREATE TABLESPACE*.

On DB2 servers that are enabled to support Lightweight Directory Access Protocol (LDAP), the new database is automatically registered in the LDAP directory.



The following example creates a database called **sales** with specifically declared *NUMSEGS*, *DFT_EXTENT_SZ*, and *AUTOCONFIGURATION* enabled and applied:

```
CREATE DATABASE sales
NUMSEGS 200 DFT_EXTENT_SZ 1000
WITH "Temporary sales database"
AUTOCONFIGURE APPLY DB ONLY;
```

The *WITH* clause adds a description of the **sales** database to the database directory.

MySQL

In MySQL, *CREATE DATABASE* essentially creates a new directory that holds the database objects:

```
CREATE DATABASE [IF NOT EXISTS] database_name
```

where:

CREATE DATABASE *database_name*

Creates a database and directory of *database_name*. The database directory appears under the MySQL data directory. Tables in MySQL then appear as files in the database directory.

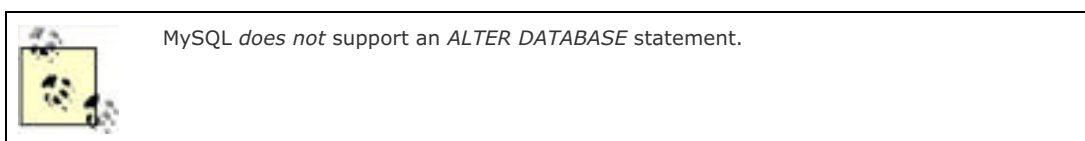
IF NOT EXISTS

Avoids an error if the database already exists.

For example, we could create a database called **sales_archive** on a MySQL server. If the database already exists, we want to abort the command without causing an error:

CREATE DATABASE IF NOT EXISTS sales_archive

Any tables that we create in the **sales_archive** database will now appear as files within the **sales_archive** directory.



Oracle

Oracle provides an extraordinary level of control over database file structures beyond merely naming the database and specifying a path for the database files. *CREATE* and *ALTER DATABASE* are very powerful commands in Oracle. Some of the more sophisticated clauses are best used only by experienced DBAs. These commands can be very large and complex. (*ALTER DATABASE* alone consumes over 50 pages in the Oracle vendor documentation!)

Novices should be aware that *CREATE DATABASE*, when run, erases all data in the specified datafiles that are already in existence. Similarly, all the data in an existing database will be lost.

Following is the syntax to create a new database in Oracle:

```
CREATE DATABASE [database_name]  
{[USER SYS IDENTIFIED BY password | USER SYSTEM IDENTIFIED BY password]}  
[CONTROLFILE REUSE]  
{[LOGFILE definition [,... ] [MAXLOGFILES int] [[MAXLOGMEMBERS] int]  
  [[MAXLOGHISTORY] int] [{ARCHIVELOG | NOARCHIVELOG}] [FORCE LOGGING]}  
[MAXDATAFILES int]  
[MAXINSTANCES int]  
[CHARACTER SET charset]  
[NATIONAL CHARACTER SET charset]  
[EXTENT MANAGEMENT {DICTIONARY | LOCAL}]  
[DATAFILE definition [,... ]]  
[SYSAUX DATAFILE definition [,... ]]  
[DEFAULT TABLESPACE tablespace_name  
  [DATAFILE file_definition]  
  EXTENT MANAGEMENT {DICTIONARY |  
    LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M] } } ]
```

```
[ [{ BIGFILE | SMALLFILE}] DEFAULT [TEMPORARY] TABLESPACE tablespace_name
    [TEMPFILE file_definition]
    EXTENT MANAGEMENT {DICTIONARY |
        LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M] ] } } ]
[ [{ BIGFILE | SMALLFILE}] UNDO TABLESPACE tablespace_name
    [DATAFILE temp_datafile_definition] ]
[SET TIME_ZONE = ' { {+ | -} hh:mi | time_zone_region}' ]
[SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE]
```

Following is the syntax to alter an existing database:

```
ALTER DATABASE [database_name]
[ARCHIVELOG | NOARCHIVELOG] |
{MOUNT [{STANDBY | CLONE} DATABASE] |
    OPEN [READ WRITE | READ ONLY
        [RESETLOGS | NORESETLOGS] | [UPGRADE | DOWNGRADE] ] } |
{ACTIVATE [PHYSICAL | LOGICAL] STANDBY DATABASE
    [SKIP [STANDBY LOGFILE] ] |
    SET STANDBY [DATABASE] TO MAXIMIZE {PROTECTION | AVAILABILITY |
        PERFORMANCE} |
    REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE ['file']
        [FOR logminer_session_name] |
    {COMMIT | PREPARE} TO SWITCHOVER TO
        {[PHYSICAL | LOGICAL] PRIMARY | STANDBY} [WITH[OUT]
            SESSION SHUTDOWN] [WAIT | NOWAIT]} |
    CANCEL} |
    START LOGICAL STANDBY APPLY [IMMEDIATE] [NODELAY]
        [{INITIAL int | NEW PRIMARY dblink_name | {FINISH |
            SKIP FAILED TRANSACTION] } ] |
    {STOP | ABORT} LOGICAL STANDBY APPLY |
    [[NO]PARALLEL int] } |
{RENAME GLOBAL_NAME TO database [.domain [.domain ...]] |
    CHARACTER SET character_set |
    NATIONAL CHARACTER SET character_set |
    DEFAULT TABLESPACE tablespace_name |
    DEFAULT TEMPORARY TABLESPACE {GROUP int | tablespace_name} |
    {DISABLE BLOCK CHANGE TRACKING | ENABLE BLOCK CHANGE TRACKING [USING
        FILE 'file' [REUSE] ] } |
    FLASHBACK {ON | OFF} |
    SET TIME_ZONE = ' { {+ | -} hh:mi | time_zone_region}' |
```

```
SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE } |
{ENABLE | DISABLE} { [PUBLIC] THREAD int | INSTANCE 'instance_name' } |
{GUARD {ALL | STANDBY | NONE} } |
{CREATE DATAFILE 'file' [,...] [AS {NEW | file_definition [,...]} ] |
  {DATAFILE 'file' | TEMPFILE 'file' [,...]}
  {ONLINE | OFFLINE [FOR DROP | RESIZE int [K | M] |
    END BACKUP | AUTOEXTEND {OFF | ON [NEXT int [K | M] [MAXSIZE
    [UNLIMITED | int [K | M] ] } |
  {TEMPFILE 'file' | TEMPFILE 'file' [,...]}
  {ONLINE | OFFLINE | DROP [INCLUDING DATAFILES] |
    RESIZE int [K | M] | AUTOEXTEND {OFF | ON [NEXT int [K | M]
    [MAXSIZE [UNLIMITED | int [K | M] ] } |
  RENAME FILE 'file' [,...] TO 'new_file_name' [,...] } |
{[[NO] FORCE LOGGING] | [ [NO]ARCHIVELOG [MANUAL] ] |
  [ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE |
  FOREIGN KEY) [,...]] COLUMNS |
[ADD | DROP] [STANDBY] LOGFILE
  {[THREAD int | INSTANCE 'instance_name']} {[GROUP int |
  logfile_name [,...]} [SIZE [K | M]] | [REUSE] |
  [MEMBER] 'file' [REUSE] [,...] TO logfile_name [,...]} |
ADD LOGFILE MEMBER 'file' [REUSE] [,...] TO {[GROUP int |
  logfile_name [,...]} |
DROP [STANDBY] LOGFILE {MEMBER 'file' | {[GROUP int | logfile_name
  [,...]} }
CLEAR [UNARCHIVED] LOGFILE {[GROUP int | logfile_name [,...]} [,...]
  [UNRECOVERABLE DATAFILE] } |
{CREATE [LOGICAL | PHYSICAL] STANDBY CONTROLFILE AS 'file' [REUSE] |
  BACKUP CONTROLFILE TO
  {'file' [REUSE] | TRACE [AS 'file' [REUSE] ] {RESETLOGS |
  NORESETLOGS}} } |
{RECOVER
  {[AUTOMATIC [FROM 'location']] |
  {[STANDBY] DATABASE
  {[UNTIL {CANCEL | TIME date | CHANGE int} |
  USING BACKUP CONTROLFILE } |
  {[STANDBY] {TABLESPACE tablespace_name [,...]} | DATAFILE
  'file' [,...]} [UNTIL [CONSISTENT WITH] CONTROLFILE]} |
  TABLESPACE tablespace_name [,...]} | DATAFILE 'file' [,...]} |
  LOGFILE filename [,...]} {[TEST | ALLOW int CORRUPTION | [NO]PARALLEL
```

```
int}} |  
CONTINUE [DEFAULT] |  
CANCEL [IMMEDIATE][NOWAIT] }  
{MANAGED STANDBY DATABASE  
{[[NO]TIMEOUT int] [{[NO]DELAY int | DEFAULT DELAY}] [NEXT int]  
[[NO]EXPIRE int] [{[NO]PARALLEL int}] [USING CURRENT LOGFILE]  
[UNTIL CHANGE int]  
[THROUGH  
{ALL ARCHIVELOG | {ALL | LAST | NEXT} SWITCHOVER |  
[[THREAD int] SEQUENCE int] } |  
[CANCEL [IMMEDIATE] [{WAIT | NOWAIT}]] |  
[DISCONNECT [FROM SESSION] [{[NO]PARALLEL int}]  
FINISH [SKIP [STANDBY LOGFILE]] [{WAIT | NOWAIT}]] |  
  
{BEGIN | END} BACKUP} }
```

The syntax elements in Oracle are as follows:

{CREATE | ALTER} DATABASE *database_name*

Creates or alters a database with the name *database_name*. The database name can be up to 8 bytes in length and may not contain European or Asian characters. You can omit a database name and allow Oracle to create a database name for you, but beware that Oracle can create some counter-intuitive names.

USER SYS IDENTIFIED BY *password*

USER SYSTEM IDENTIFIED BY *password*

Specifies passwords for the *SYS* and *SYSTEM* users. You may specify neither or both of these clauses, but not just one of them.

CONTROLFILE REUSE

Causes existing control files to be reused, enabling you to specify existing files in the *CONTROL_FILES* parameter in *INIT.ORA*. Oracle will then overwrite any information those files may contain. This clause is normally used when recreating a database. Consequently, you probably don't want to use this clause in conjunction with *MAXLOGFILES*, *MAXLOGMEMBER*, *MAXLOGHISTORY*, *MAXDATAFILES*, or *MAXINSTANCES*.

LOGFILE *definition*

Specifies one or more logfiles for the database. You may define multiple files all with the same size and characteristics, using the *file* parameter, or you may define multiple files, each with its own size and characteristics. The entire logfile definition syntax is rather ponderous, but offers a great deal of control:

```
LOGFILE { ('file' [...]) [SIZE int [K | M]
```

```
[GROUP int] [REUSE]} [...]
```

where:

LOGFILE ('*file*' [,...])

Defines one or more files that will act as redo logfiles; *file* is both the filename and path. Any files defined in the *CREATE DATABASE* statement are assigned to redo log thread number 1. When specifying multiple redo logfiles, each filename should be enclosed in single quotes and separated from the other names by commas. The entire list should be enclosed in parentheses.

SIZE *int* [K | M]

Specifies the size of the redo logfile in bytes as an integer value, *int*. Alternately, you may define the redo logfile in large units than bytes by appending a K (for kilobytes) or an M (for megabytes).

GROUP *int*

Defines the integer ID, *int*, of the redo logfile group. The value may be from 1 to the value of the *MAXLOGFILES* clause. An Oracle database must have at least two redo logfile groups. Oracle will create a redo logfile group for you, at 100 MB each, if you leave off this group ID.

REUSE

Reuses an existing redo logfile.

MAXLOGFILES *int*

Sets the maximum number of logfiles, *int*, available to the database being created. The minimum, maximum, and default for this clause are OS-dependent.

MAXLOGMEMBERS *int*

Sets the maximum number of members (i.e., copies) for a redo logfile group. The minimum value is 1 while the maximum and default values for this clause are OS-dependent.

MAXLOGHISTORY *int*

Sets the maximum number of archived redo logfiles available to a Real Application Cluster. You can use the *MAXLOGHISTORY* clause only when Oracle is in *ARCHIVELOG* mode on a Real Application Cluster. The minimum value is 0 while the maximum and default values for this clause are OS-dependent.

ARCHIVELOG | NOARCHIVELOG

Defines how redo logs operate. When used with *ALTER DATABASE*, allows the current setting to be changed. *ARCHIVELOG* saves data stored in the redo log(s) to an archiving file, providing for media recoverability. Conversely, *NOARCHIVELOG* allows a redo log to be reused without archiving the contents. Both options provide recoverability, although *NOARCHIVELOG* (the default) does not provide media recovery.

FORCE LOGGING

Places all instances of the database into *FORCE LOGGING* mode, in which all changes to the database are logged, except for changes to temporary tablespaces and segments. This setting takes precedence over any tablespace- or object-level settings.

MAXDATAFILES *int*

Sets the initial number of datafiles, *int*, available to the database being created. Note that the *INIT.ORA* setting, **DB_FILES**, also limits the number of datafiles accessible to the database instance.

MAXINSTANCES *int*

Sets the maximum number of instances, *int*, which may mount and open the database being created. The minimum value is 1 while the maximum and default values for this clause are OS-dependent.

CHARACTER SET *charset*

Controls the language character set in which the data is stored. The value for *charset* cannot be AL16UTF16. The value of *charset* is OS-dependent.

NATIONAL CHARACTER SET *charset*

Controls the national language character set for data stored in *NCHAR*, *NCLOB*, and *NVARCHAR2* columns. The value for *charset* must be either AL16UTF16 (the default) or UTF8.

EXTENT MANAGEMENT {*DICTIONARY* | *LOCAL*}

Creates a locally managed *SYSTEM* tablespace, otherwise the *SYSTEM* tablespace will be dictionary managed. This clause requires a default temporary tablespace. If you omit a *DATAFILE* clause, you can also omit a default temporary tablespace because Oracle will create them both for you.

DATAFILE *definition*

Specifies one or more datafiles for the database. (All these datafiles become part of the *SYSTEM* tablespace.) You may repeat filenames to define multiple files all with the same size and characteristics. Alternately, you may repeat the entire *DATAFILE* clause, with each occurrence defining one or more files with the same size and characteristics. The entire datafile definition syntax is rather large, but offers a great deal of control:

```
DATAFILE {'file1' [,...]} [GROUP int] [SIZE int [K | M] [REUSE]
[AUTOEXTEND [OFF | ON [NEXT int [K | M]
[MAXSIZE [UNLIMITED | int [K | M]]] } [,...]
```

where:

```
DATAFILE ('file1' [,...])
```

Defines one or more files that will act as the datafile(s), where *file1* is both the filename and path. For multiple files, each filename should be enclosed in single-quotes and separated by a comma. The entire list should be enclosed in parentheses.

```
GROUP int
```

Defines the integer ID, *int*, of the datafile group. The value may be from 1 to the value of *MAXLOGFILES* clause. An Oracle database must have at least two datafile groups. Oracle will create them for you, at 100 MB each, if you leave off this clause.

```
SIZE int [K | M]
```

Specifies the size of the datafile in bytes as an integer value, *int*. Alternately, you may define the datafile in large units by appending a K (for kilobytes) or an M (for megabytes).

```
REUSE
```

Reuses the existing datafile.

```
AUTOEXTEND
```

Enables (*ON*) the automatic extension of new or existing data or temp files (but does not redo logfiles). *NEXT* specifies the next increment of space allocated to the file in bytes, kilobytes (*K*), or megabytes (*M*) when more space is needed.

```
MAXSIZE [UNLIMITED | int [K | M]]
```

Specifies the maximum disk space allowed for automatic extension of the file. *UNLIMITED* allows the file to grow without an upper limit, except of course the total capacity of the drive. Otherwise, you may define the maximum size limit as an integer, *int*, in bytes (the default), kilobytes with the keyword *K*, and megabytes with the keyword *M*.

SYSAUX DATAFILE *definition*

Specifies one or more datafiles for the *SYSAUX* tablespace. By default, Oracle creates and manages the *SYSTEM* and *SYSAUX* tablespaces automatically. You must use this clause if you have specified a datafile for the *SYSTEM* tablespace. If you omit the *SYSAUX* clause when using Oracle-managed files, then Oracle will create the *SYSAUX* tablespace as an online, permanent, locally managed tablespace with a single datafile of 100 MB, using automatic segment-space management and logging.

BIGFILE | *SMALLFILE*

Specifies the default file type of a subsequently created tablespace. *BIGFILE* indicating the tablespace will contain a single datafile or tempfile of up to 8 exabytes (8 million terabytes) in size, while *SMALLFILE* indicates the tablespace is a traditional Oracle tablespace. The default, when omitted, is *SMALLFILE*.

DEFAULT TABLESPACE *tablespace_name*

Specifies a default permanent tablespace for the database for all non-*SYSTEM* users. When this clause is omitted, the *SYSTEM* tablespace is the default permanent tablespace for non-*SYSTEM* users. The *EXTENT MANAGEMENT* clause has the same syntax as in the *DEFAULT TEMPORARY TABLESPACE* clause appearing later.

DEFAULT TEMPORARY TABLESPACE *tablespace_name* [TEMPFILE *file_definition*]

Defines the name and location of the default temporary tablespace for the database. Users who are not explicitly assigned to a temporary tablespace will operate in this one. If you don't create a default temporary tablespace, Oracle uses the *SYSTEM* tablespace. Under *ALTER DATABASE*, this clause allows you to change the default temporary tablespace.

TEMPFILE *file_definition*

The tempfile definition is optional when the *DB_CREATE_FILE_DEST INIT.ORA* parameter is set. Otherwise, you'll have to define the tempfile yourself. The *TEMPFILE* definition syntax is otherwise identical to the *DATAFILE* definition syntax earlier in this section.

EXTENT MANAGEMENT {DICTIONARY | LOCAL {AUTOALLOCATE | UNIFORM [SIZE *int* [K | M]}}

Defines the way in which the *SYSTEM* tablespace is managed. When this clause is omitted, the *SYSTEM* tablespace is dictionary-managed. Once created as a locally managed tablespace, it cannot be converted back to a dictionary-managed tablespace, nor can any new dictionary-managed tablespaces be created in the database.

DICTIONARY

Specifies that the Oracle data dictionary manages the tablespace. This is the default. The *AUTOALLOCATE* and *UNIFORM* subclauses are not used with this clause.

LOCAL

Declares that the tablespace is locally managed. This clause is optional since all temporary tablespaces have locally managed extents by default. Use of this clause requires a default temporary tablespace. If you do not manually create one, Oracle will automatically create one called *TEMP* of 10 MB size with autoextend disabled.

AUTOALLOCATE

Specifies that new extents will be allocated as needed by the locally managed tablespace.

UNIFORM [SIZE *int* [K | M]]

Specifies that all extents of the tablespace are the same size (*UNIFORM*), in bytes, as *int*. The *SIZE* clause allows you to configure the size of the extents to your liking in bytes (the default), in kilobytes using the keyword *K*, or in megabytes using the keyword *M*. The default is 1M.

UNDO TABLESPACE *tablespace_name* [DATAFILE *temp_datafile_definition*]

Defines the name and location for undo data, creating a tablespace named *tablespace_name*, but only if you have set the *UNDO_MANAGEMENT INIT.ORA* parameter to *AUTO*. If you don't use this clause, Oracle manages undo space via rollback segments. (You may also set the *INIT.ORA* parameter to *UNDO_TABLESPACE*. If so, the value of the parameter and the *tablespace_name* used here must be identical.)

where:

DATAFILE *temp_datafile_definition*

Creates and assigns the datafile, as you have defined it, to the undo tablespace. Refer to the earlier section on *DATAFILE* for the full syntax on this clause. This clause is required if you have not specified a value for the *INIT.ORA* parameter *DB_CREATE_FILE_DEST*.

SET TIME_ZONE = ' { {+ | -} *hh:mi* | *time_zone_region* }'

Sets the time zone for the database, either by specifying a delta from Greenwich Mean Time (now called Coordinated Universal Time) or by specifying a time zone region. (For a list of time zone regions, query the **tzname** column of the *v\$timezone_names* view.) If you do not use this clause, Oracle defaults to the operating-system time zone.

SET DEFAULT TABLESPACE

Sets all tablespaces created by the current *CREATE DATABASE* or *ALTER DATABASE* statement as either *BIGFILE* or *SMALLFILE*. When creating databases, this clause also applies to the *SYSTEM* and *SYSAUX* tablespaces.

MOUNT [{*STANDBY*} | *CLONE*] DATABASE]

Mounts a database for users to access. The *STANDBY* keyword mounts a physical standby database, enabling it to receive archived redo logs from the primary instance. The *CLONE* keyword mounts a clone database. This clause cannot be used with *OPEN*.

OPEN [*READ WRITE* | *READ ONLY*] [*RESETLOGS* | *NORESETLOGS*] [*UPGRADE* | *DOWNGRADE*]

Opens the database separately from the mounting process. (Mount the database first.) *READ WRITE* opens the database in read/write mode, allowing users to generate redo logs. *READ ONLY* allows reads but disallows changes to redo logs. *RESETLOGS* discards all redo information not applied during recovery and sets the log sequence number to 1. *NORESETLOGS* retains the logs in their present condition. The optional *UPGRADE* and *DOWNGRADE* clauses tell Oracle to dynamically modify the system parameters as required for database upgrade or downgrade, respectively. The default is *OPEN READWRITE NORESETLOGS*.

ACTIVATE [*PHYSICAL* | *LOGICAL*] STANDBY DATABASE [*SKIP* [*STANDBY LOGFILE*]]

Promotes a standby database to the primary database. You can optionally specify a *PHYSICAL* standby, the default, or a *LOGICAL* standby. Use the *SKIP* clause to immediately promote a physical standby and discard any data still unapplied by the *RECOVER MANAGED STANDBY DATABASE FINISH* statement. The clause *STANDBY LOGFILE* is noise.

SET STANDBY [*DATABASE*] TO MAXIMIZE {*PROTECTION* | *AVAILABILITY* | *PERFORMANCE*}

Sets the level of protection for data in the primary database. The old terms *PROTECTED* and *UNPROTECTED* equate to *TO MAXIMIZE PROTECTION* and *MAXIMIZE PERFORMANCE*, respectively, where:

PROTECTION

Provides the highest level of data protection, but has the greater overhead and negatively impacts availability. This setting commits transactions only after all data necessary for recovery has been physically written in at least one physical standby database that uses *SYNC* log transport mode.

AVAILABILITY

Provides the second highest level of data protection, but the highest level of availability. This setting commits transactions only after all data necessary for recovery has been physically written in at least one physical or logical standby database that uses SYNC log transport mode.

PERFORMANCE

Provides the highest level of performance, but compromises data protection and availability. This setting commits transactions before all data necessary for recovery has been physically written to a standby database.

REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE [*file*]

Manually registers redo logfiles from a failed primary server when issued from a standby server. The logfile may optionally be declared as *PHYSICAL* or *LOGICAL*. The *OR REPLACE* clause allows updates to details for an existing archivelog entry.

FOR *logminer_session_name*

Registers the logfile with a single, specific LogMiner session in an Oracle Streams environment.

{*COMMIT* | *PREPARE*} TO SWITCHOVER TO {[*PHYSICAL* | *LOGICAL*] PRIMARY | STANDBY}

Performs a graceful switchover, moving the current primary database to standby status and promoting a standby database to primary. (In a RAC environment, all instances other than the current instance have to be shutdown.) To gracefully switch over, you should issue the command twice, once to prepare the primary and standby databases to begin exchanging logfiles in advance of the switchover using *PREPARE TO SWITCHOVER*. To demote the primary database and switchover of the standby databases, use *COMMIT TO SWITCHOVER*. The *PHYSICAL* clause puts the primary database into physical standby mode. The *LOGICAL* clause puts the primary database into logical standby mode. However, you must then issue an *ALTER DATABASE START LOGICAL STANDBY APPLY* statement.

WITH[OUT] SESSION SHUTDOWN] [WAIT | NOWAIT]

WITH SESSION SHUTDOWN closes any open application sessions and rolls back any uncommitted transactions during a switchover of physical databases (but not logical ones). *WITHOUT SESSION SHUTDOWN*, the default, causes the *COMMIT TO SWITCHOVER* statement to fail if it encounters any open application sessions. *WAIT* returns control to the console after completion of the *SWITCHOVER* command, while *NOWAIT* returns control before the command completes.

START LOGICAL STANDBY APPLY [IMMEDIATE][NODELAY][{INITIAL *int* | NEW PRIMARY *dblink_name*} | {FINISH | SKIP} FAILED TRANSACTIONS}]

Starts to apply redo logs to the logical standby database. *IMMEDIATE* tells the Oracle LogMiner to read the redo data in the standby redo logfiles. *NODELAY* tells Oracle to ignore a delay for the apply, such as when the primary database is unavailable or disabled. *INITIAL* is used the first time you apply logs to the standby database. *NEW PRIMARY* is required after a switchover has completed or after a standby database has processed all redo logs and another standby is promoted to primary. Use *SKIP FAILED TRANSACTION* to skip the last transaction and to restart the apply. Use *FINISH* to apply the data in the redo logs if the primary database is disabled.

[STOP | ABORT] LOGICAL STANDBY APPLY

Stops the application of redo logs to a logical standby server. *STOP* performs an orderly stop, while *ABORT* performs an immediate stop.

[NO]PARALLEL [*int*]

[*NO*]PARALLEL determines whether parallel recovery of media is used. *NOPARALLEL* is the default and enforces serial reading of the media. *PARALLEL* with no *int* value tells Oracle to choose the degree of parallelism to apply. Specifying *int* declares the degree of parallelism to apply.

RENAME GLOBAL_NAME TO *database*[.*domain*[.*domain* ...]]

Changes the global name of the database, where *database* is the new name of up to 8 bytes in length. The optional *domain* specifications identify the database's location in the network. This does not propagate database name changes to any dependent objects like synonyms, stored procedures, etc.

{DISABLE | ENABLE} BLOCK CHANGE TRACKING [USING FILE '*file*'] [REUSE]

Tells Oracle to stop or start tracking the physical location of all database updates, respectively, and maintain the information in a special file called the block change tracking file. Oracle will automatically create the file as defined by the *DB_CREATE_FILE_DEST* parameter, unless you add the *USING FILE 'file'* clause, where '*file*' is the path and name of the file. *REUSE* tells Oracle to overwrite an existing block change tracking file of the same name as '*file*'. The *USING* and *REUSE* subclauses are only allowed with the *ENABLE BLOCK* clause.

FLASHBACK {ON | OFF}

Places the database in or out of *FLASHBACK* mode, respectively. When in flashback mode, an Oracle database automatically creates and maintains flashback database logs in the flash recovery area. When *OFF*, the flashback database logs are deleted and unavailable.

SET TIME_ZONE

Specifies the time zone for the server. Refer to the *SET TIME_ZONE* statement for more information.

ENABLE [PUBLIC] THREAD *int* | DISABLE THREAD *int* | INSTANCE '*instance_name*'

In Real Application Cluster (RAC) environments, you can *ENABLE* or *DISABLE* a redo log thread by number (*int*). You may optionally specify an *instance_name* to enable or disable a thread mapped to a specific database instance of an Oracle Real Application Cluster environment. The *instance_name* may be up to 80 characters long. The *PUBLIC* keyword makes the thread available to any instance. When omitted, the thread is only available when explicitly requested. To enable a thread, the thread must have at least two redo logfile groups. To disable a thread, the database must be open but not mounted by an instance using the thread.

GUARD {ALL | STANDBY | NONE}

Protects the data in a database from changes. *ALL* prevents users other than SYS from making changes. *STANDBY* prevents all users other than SYS from making changes in a logical standby. *NONE* provides normal security for the database.

CREATE DATAFILE '*file*' [,...] [AS {NEW | *file_definition*}]

Creates a new empty datafile, replacing an existing one. The value '*file*' identifies a file (either by filename or file number) that was lost or damaged without a backup. *AS NEW* creates a new file in the default filesystem using an Oracle-supplied name. *AS file_definition* allows you to specify a filename and sizing details, as defined under the *file_definition* section earlier in the list.

DATAFILE '*file*' | TEMPFILE '*file*' [,...] {ONLINE | OFFLINE [FOR DROP]}
| RESIZE *int* [K | M] | AUTOEXTEND {OFF | ON [NEXT
int [K | M]]} [MAXSIZE [UNLIMITED | *int* [K | M]]] {END BACKUP [INCLUDING DATAFILES]}

Changes the attributes, such as the size, of one or more existing datafiles or tempfiles. You may alter one or more files in a comma-delimited list, identified in the value '*file*' by filename or file number. Do not mix datafile and tempfile declarations; only one or the other should appear in this clause at a time, where:

ONLINE

Sets the file online.

OFFLINE [FOR DROP]

Sets the file offline allowing media recovery. *FOR DROP* is required to take a file offline in *NOARCHIVELOG* mode, but does not actually destroy the file. It is ignored in *ARCHIVELOG* mode.

RESIZE *int* [K | M]

Sets a new size for an existing datafile or tempfile.

END BACKUP

Described later in the list under *END BACKUP*. Used only with the *DATAFILE* clause.

DROP INCLUDING DATAFILES

Drops not only the tempfile, but all datafiles on the filesystem associated with the tempfile. Oracle also adds an entry to the alert log for each file that is erased. Used only with the *TEMPFILE* clause.

AUTOEXTEND {OFF | ON [NEXT *int* [K | M] [MAXSIZE [UNLIMITED | *int* [K | M]]}

Described earlier in the list under *DATAFILE* definition.

RENAME FILE '*file*' [,...] TO '*new_file_name*' [,...]

Renames a datafile, tempfile, or redo logfile member from the old name, '*file*', to the '*new_file_name*'. You can rename multiple files at once by specifying multiple old and new filenames. This command does not rename files at the operating system level. Rather, it specifies new names that Oracle will use to open the files. You need to rename at the operating system level yourself.

[NO] FORCE LOGGING

Puts the database into force logging mode (*FORCE LOGGING*) or takes it out of force logging mode (*NO FORCE LOGGING*). In the former, Oracle logs all changes to the database except in temporary tablespaces or segments. This database-level *FORCE LOGGING* setting supersedes all tablespace level declarations regarding force logging mode.

[NO]ARCHIVELOG [MANUAL]

Tells Oracle to create redo logfiles, but the user will handle the archiving of the redo logfiles explicitly. It is used only with the *ALTER DATABASE* statement and only for backward compatibility for users with older tape backup systems. When this clause is omitted, Oracle defaults the redo logfile destination to the *LOG_ARCHIVE_DEST_n* initialization parameter.

[ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY) [,...] COLUMNS]

ADD places additional column data into the log stream whenever an update is executed. It also enables *minimal supplemental logging*, which ensures that LogMiner can support chained rows and special storage arrangements like clustered tables. Supplemental logging is disabled by default. You can add the clauses *PRIMARY KEY COLUMNS*, *UNIQUE KEY COLUMNS*, *FOREIGN KEY COLUMNS*, or *ALL* (to get all three earlier options), if you need to enable full referential integrity via foreign keys in another database, such as a logical standby. In either case, Oracle places either the primary key columns, unique key columns (or where none exist, a combination of columns that uniquely identify each row), foreign key columns, or all three into the log. *DROP* tells Oracle to suspend supplemental logging.

[ADD | DROP] [STANDBY] LOGFILE [THREAD *int*] [GROUP *int*] [MEMBER] '*file*' [,...]

ADD adds one or more primary or standby redo logfile groups to the specified instance. *THREAD* assigns the added files to a specific thread number (*int*) on a Real Application Cluster. When omitted, the default is the thread assigned to the current instance. *GROUP* assigns the redo logfile groups to a specific group within the thread. *MEMBER* adds the specified '*file*' (or files in a comma-delimited list) to an existing redo logfile group. *DROP LOGFILE MEMBER* drops one or more redo logfile members, after issuing an *ALTER SYSTEM SWITCH LOGFILE* statement.

CLEAR [UNARCHIVED] LOGFILE GROUP *int logfile_name* [,...] [UNRECOVERABLE DATAFILE] }

Reinitializes one or more specified online redo logs. Specify the logs in a comma-delimited list. *UNARCHIVED* skips archiving the redo logs. *UNRECOVERABLE DATAFILE* is required when any datafile is offline and the

database is in *ARCHIVELOG* mode.

CREATE {LOGICAL | PHYSICAL} STANDBY CONTROLFILE AS '*file*' [REUSE]}

Creates a control file that maintains a logical or physical standby database. *REUSE* is needed if the file already exists.

BACKUP CONTROLFILE TO {'*file*' [REUSE] | TRACE [AS '*file*' [REUSE]] [RESETLOGS | NORESETLOGS]}

Backs up the current control file of an open or mounted database. *TO 'file'* identifies a full path and filename for the control file. *TO TRACE* writes SQL statements to recreate the control file to a trace file. *TO TRACE AS 'file'* writes all the SQL statements to a standard file rather than a trace file. *REUSE* is needed if the file already exists. *RESETLOGS* initializes the trace file with the statement *ALTER DATABASE OPEN RESETLOGS*, and is only valid when online logs are unavailable. *NORESETLOGS* initializes the trace file with the statement *ALTER DATABASE OPEN NORESETLOGS* and is only valid when online logs are available.

RECOVER

Controls media recovery for the database, standby database, tablespace, or file. In Oracle, the *ALTER TABLE* command is one of the primary means of recovering a damaged or disabled database, file, or tablespace. Use *RECOVER* when the database is mounted (in exclusive mode), the files and tablespaces involved are not in use (offline), and the database is in either an open or closed state. The entire database can be recovered only when it is closed, but specific files or tablespaces can be recovered in a database that is open.

AUTOMATIC [FROM '*location*']

Tells Oracle to automatically generate the name of the next archived redo logfile necessary for continued operation during recovery. Oracle will prompt you if it cannot find the next file. The *FROM 'location'* clause tells Oracle where to find the archived redo logfile group. The following subclauses may be applied to an automatically recovered database.

STANDBY

Specifies that the database or tablespace to recover is a standby type.

DATABASE {UNTIL {CANCEL | TIME *date* | CHANGE *int*} | USING BACKUP CONTROLFILE }

Tells Oracle to recover the entire database. The *UNTIL* keyword tells Oracle to continue recovery until *ALTER DATABASE...RECOVER CANCEL (CANCEL)* is issued, until a specified time is reached (*TIME*) in the format YYYY-MM-DD:HH24:MI:SS, or until a specific system change number (*CHANGE int*) is reached where *int* is the number. The clause *USING BACKUP CONTROLFILE* enables use of the backup, rather than current, control file.

[STANDBY] TABLESPACE *tablespace_name* [,...] | DATAFILE '*file*' [,...]

Recovers one or more specific tablespaces or datafiles, respectively. You may specify more than one tablespace or datafile using a comma-delimited list. You may also recover a datafile by datafile number, rather than by name. The tablespace may be in normal or standby mode. The standby tablespace or datafile is reconstructed using archived redo logfiles copied from the primary database and a control file.

UNTIL [CONSISTENT WITH] CONTROLFILE

Tells Oracle to recover an older standby tablespace or datafile by using the current standby control file. *CONSISTENT WITH* are noise words.

LOGFILE *filename* [,...]

Continues media recovery by applying one or more redo logfiles that you specify in a comma-delimited list.

TEST | ALLOW *int* CORRUPTION | [NO]PARALLEL *int*

TEST performs a trial recovery, allowing you to foresee any problems. *ALLOW int CORRUPTION* tells how many corrupt blocks (*int*) to tolerate before causing recovery to abort. *int* must be 1 for a real recovery, but may be any number you chose when paired with *TEST*. Parallelism is described earlier in the list.

CONTINUE [DEFAULT]

Determines whether multi-instance recovery continues after interruption. *CONTINUE DEFAULT* is the same as *RECOVER AUTOMATIC*, but does not result in a prompt for a filename.

CANCEL [IMMEDIATE][[NO]WAIT]

Cancels a managed recovery operation at the next archived log boundary, if it was started with the *USING CANCEL* clause. The *IMMEDIATE* keyword cancels the managed recovery in process after the current I/O or archived log boundary, whichever is first. *NOWAIT* cancels the recovery immediately without regard to current work in process, while *WAIT* allows current work to complete normally.

MANAGED STANDBY DATABASE

Specifies managed physical standby recovery mode on an active component of a standby database. This command is used for media recovery only and not to construct a new database, using the following parameters:

[NO]TIMEOUT int

TIMEOUT specifies, in minutes (*int*), how long the foreground managed recovery operation will wait for the next available redo log before timing out. *NOTIMEOUT*, the default, tells Oracle to wait indefinitely for the next redo log, or until the instance is shutdown, or until *RECOVER CANCEL* is issued.

CANCEL [IMMEDIATE][[NO]WAIT]

Described earlier.

[NO]DELAY int | DEFAULT DELAY int

Tells, in minutes (*int*), how long Oracle should wait to apply an archived redo log after it has been selected for recovery. *DEFAULT DELAY* tells Oracle to use the delay specified by the *LOG_ARCHIVE_DEST_n INIT.ORA* parameter. *NODELAY* tells Oracle to apply the archived redo log immediately.

NEXT int

Tells Oracle to apply the *int* next archived redo logs as soon as possible after they have been archived. Incompatible with *FINISH*.

[NO]EXPIRE int

Tells Oracle, in minutes (*int*), when to automatically terminate a managed recovery relative to the time the recovery started. *NOEXPIRE* disables a previously set *EXPIRE* value. Incompatible with *FINISH*.

USING CURRENT LOGFILE

Invokes *real time apply*, which allows recovery of redos from standby online logs as they are being filled, without first requiring that they be archived by the standby database.

UNTIL CHANGE int

Conducts a managed recovery up to (but not including) the specified system change number called *int*.

THROUGH {ALL ARCHIVELOG | {ALL | LAST | NEXT} SWITCHOVER | [THREAD int] SEQUENCE int}

Tells Oracle when to terminate managed recovery to a standby server. *THROUGH ALL ARCHIVELOG*, the default, continues managed recovery until all archive logs are recovered. *THROUGH...SWITCHOVER* controls the termination of managed recovery when Oracle switches over to a secondary standby server. *ALL* keeps managed recovery in operation through any switchovers. *LAST* terminates managed recovery after receiving the final end-of-redo indicator. *NEXT* terminates managed recovery after

receiving the next end-of-redo indicator. *THROUGH...SEQUENCE* tells Oracle to terminate managed recovery when it encounters a specific *THREAD* number and/or *SEQUENCE* number.

DISCONNECT [FROM SESSION] [[NO]PARALLEL *int*]

Causes the managed redo process to occur in the background, leaving the current process available for other tasks. *FROM SESSION* are noise words. *DISCONNECT* is incompatible with *TIMEOUT*.

FINISH [SKIP [STANDBY LOGFILE]] [{WAIT | NOWAIT}]

Recovers the current physical standby online redo logfiles in preparation of the standby assuming the primary database role when the primary database has failed. The *FINISH* clause, when used without any other keywords, is known as a *terminal recovery*. The optional clause *SKIP [STANDBY LOGFILE]* allows the standby to prepare for the primary role when standby online redo logfiles are not available. The *FINISH SKIP* clause, when used without any other keywords, is known as a *terminal incomplete recovery*. *WAIT* tells Oracle to return control after completing the recovery process, while *NOWAIT* returns control immediately. Incompatible with *TIMEOUT*, *DELAY*, *EXPIRE*, and *NEXT*.

{*BEGIN* | *END*} *BACKUP*

Controls online backup mode for any datafiles. *BEGIN* places all datafiles into online backup mode. The database must be mounted and open, in archivelog mode with media recovery enabled. (Note that while the database is in online backup mode the instance cannot be shutdown, and individual tablespaces cannot be backed up, taken offline, or made read only.) *END* takes all datafiles currently in online backup mode out of that mode. The database must be mounted, but need not be open.

After that long discussion of specific syntax, it's important to establish some Oracle basics.

Oracle allows the use of *primary* and *standby* databases. A *primary* database is a mounted and open database accessible to users. The primary database then ships its redo logs to a *standby* database where they are recovered, thus making the standby database a very up-to-date copy of the primary.

Unique to the Oracle environment is the *INIT.ORA* file, which specifies the database name and a variety of other options when creating and starting up the database. You should always define startup parameters, such as the name of any control files, in the *INIT.ORA* file to identify the control files or else the database will not start. Starting in Oracle 9.1, you can use binary parameter files rather than *INIT.ORA* files.

When a group of logfiles is listed, they are usually shown in parentheses. The parentheses aren't needed when creating a group with only one member, but this is seldom done. Here's an example using a parenthetical list of logfiles:

```
CREATE DATABASE publications
```

```
LOGFILE ('/s01/oradata/loga01','/s01/oradata/loga02') SIZE 5M
```

```
DATAFILE;
```

The example above creates a database called **publications** with an explicitly defined logfile clause and an automatically created datafile. The following example of an Oracle *CREATE DATABASE* command is much more sophisticated:

```
CREATE DATABASE sales_reporting
```

```
CONTROLFILE REUSE
```

```
LOGFILE
```

```
GROUP 1 ('diskE:log01.log', 'diskF:log01.log') SIZE 15M,
```

```
GROUP 2 ('diskE:log02.log', 'diskF:log02.log') SIZE 15M
```

```
MAXLOGFILES 5
```

```
MAXLOGHISTORY 100
```

```
MAXDATAFILES 10  
MAXINSTANCES 2  
ARCHIVELOG  
CHARACTER SET AL32UTF8  
NATIONAL CHARACTER SET AL16UTF16  
DATAFILE  
  'diskE:sales_rpt1.dbf' AUTOEXTEND ON,  
  'diskF:sales_rpt2.dbf' AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED  
DEFAULT TEMPORARY TABLESPACE temp_tblspc  
UNDO TABLESPACE undo_tblspc  
SET TIME_ZONE = '-08:00';
```

This example defines log and data files, as well as all appropriate character sets. We also define a few characteristics for the database, such as the use of *ARCHIVELOG* mode and *CONTROLFILE REUSE* mode, the time zone, maximum number of instances, datafiles, etc. The example also assumes that the *INIT.ORA* parameter for *DB_CREATE_FILE_DEST* has already been set. Thus, we don't have to define file definitions for the *DEFAULT TEMPORARY TABLESPACE* and *UNDO TABLESPACE* clauses.

When issued by a user with SYSDBA privileges, this statement creates a database and makes it available to users in either exclusive or parallel mode, as defined by the value of the *CLUSTER_DATABASE* initialization parameter. Any data that exists in predefined datafiles is erased. You will usually want to create tablespaces and rollback segments for the database. (Refer to the vendor documentation for the platform specific commands *CREATE TABLESPACE* and *CREATE ROLLBACK SEGMENT*.)

Oracle has tightened up security around default database user accounts. Many default database user accounts are now locked and expired during initial installation. Only *SYS*, *SYSTEM*, *SCOTT*, *DBSNMP*, *OUTLN*, *AURORA\$JIS\$UTILITY\$*, *AURORA\$ORB\$UNAUTHENTICATED*, and *OSE\$HTTP\$ADMIN* are the same as they were in earlier versions. You must manually unlock and assign a new password to all locked accounts, as well as assign a password to *SYS* and *SYSTEM* during the initial installation.

In the next example, we add more logfiles to the current database, then add a datafile:

```
ALTER DATABASE ADD LOGFILE GROUP 3  
  ('diskf: log3.sales_arch_log','diskg:log3.sales_arch_log')  
SIZE 50M;  
ALTER DATABASE sales_archive  
CREATE DATAFILE 'diskF:sales_rpt4.dbf'  
AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED;
```

We can set a new default temporary tablespace, as shown in the next example:

```
ALTER DATABASE DEFAULT TEMPORARY TABLESPACE sales_tbl_spc_2;
```

Next, we'll perform a simple full database recovery:

```
ALTER DATABASE sales_archive RECOVER AUTOMATIC DATABASE;
```

In the next example, we perform a more elaborate partial database recovery:

```
ALTER DATABASE RECOVER STANDBY DATAFILE 'diskF:sales_rpt4.dbf'  
UNTIL CONTROLFILE;
```

Now, we'll perform a simple recovery of a standby database in managed standby recovery mode:

```
ALTER DATABASE RECOVER sales_archive MANAGED STANDBY DATABASE;
```

In the following example, we gracefully switch over from a primary database to a logical standby, and promote the logical standby to primary:

```
-- Demotes the current primary to logical standby database.
```

```
ALTER DATABASE COMMIT TO SWITCHOVER TO LOGICAL STANDBY;
```

```
-- Applies changes to the new standby.
```

```
ALTER DATABASE START LOGICAL STANDBY APPLY;
```

```
-- Promotes the current standby to primary database.
```

```
ALTER DATABASE COMMIT TO SWITCHOVER TO PRIMARY;
```

PostgreSQL

PostgreSQL's implementation of the *CREATE DATABASE* command creates a database and a file location for the datafiles:

```
CREATE DATABASE database_name  
  [ WITH {[LOCATION = 'dbpath']} [TEMPLATE = tmp_name]  
  [ENCODING = enc_value]} ]
```

Where:

*WITH LOCATION = '*dbpath*'*

Specifies the file path where the database will be located. You can leave off this clause to accept the default location or use the clause *WITH LOCATION = DEFAULT*. The path must be prepared in advance using the *initlocation* command (which is a Unix command, not a SQL statement). In general, PostgreSQL uses environmental variable names (an environmental variable name is a pathname without slashes). If you want to specify an absolute file and path name, like */usr/local/pgsql/data01*, you must set and compile the server with the *ALLOW_ABSOLUTE_DBPATH* parameter.

TEMPLATE = tmp_name

Names a template to use for creating the new database. You can leave off this clause to accept the default template describing the location of the database files or use the clause *TEMPLATE = DEFAULT*. The default is to copy the standard system database *template1*. You can get a pristine database (one that contains only required database objects) by specifying *TEMPLATE = template0*.

ENCODING = enc_value

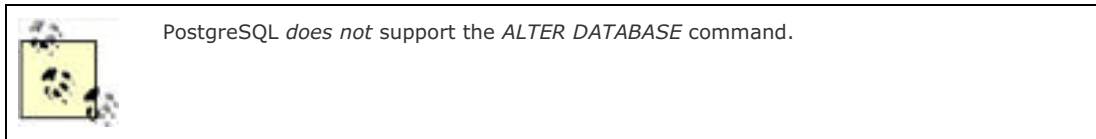
Specifies the multibyte encoding method to use in the new database using either a string literal (such as *'SQL_ASCII'*), an integer encoding number, or *DEFAULT* for the default encoding.

For example, to create the database **sales_revenue** in the */home/teddy/private_db* directory:

```
CREATE DATABASE sales_revenue WITH LOCATION =  
'/home/teddy/private_db';
```

Be careful using absolute file- and pathnames, since they impact security and data integrity issues.

PostgreSQL advises that databases used as templates be treated as read-only. Template databases should not be used for general purpose "copy database" functionality.



SQL Server

SQL Server offers a high degree of control over the OS filesystem structures that hold the database and its objects. The syntax for Microsoft SQL Server's *CREATE DATABASE* statement looks like this:

```
CREATE DATABASE database_name  
  
[ ON file_definition [...] ]  
  
[ , FILEGROUP filegroup_name file_definition [...] ]  
  
[ LOG ON file_definition [...] ]  
  
[ COLLATE collation_name ]  
  
[ FOR LOAD | FOR ATTACH ]
```

Following is the syntax for *ALTER DATABASE*:

```
ALTER DATABASE database_name  
  
{ADD FILE file_definition [...] [TO FILEGROUP filegroup_name]  
| ADD LOG FILE file_definition [...]  
| REMOVE FILE file_name  
| ADD FILEGROUP filegroup_name  
| REMOVE FILEGROUP filegroup_name  
| MODIFY FILE file_definition  
| MODIFY NAME = new_database_name  
| MODIFY FILEGROUP filegroup_name {filegroup_property  
| NAME = new_filegroup_name }  
| SET {state_option | cursor_option  
| auto_option | sql_option |  
recovery_option}  
[,...] [ WITH termination_option ]  
| COLLATE collation_name }
```

Parameter descriptions are as follows:

```
{CREATE | ALTER} DATABASE database_name
```

Creates a database, or alters an existing database, of *database_name*. The name cannot be longer than 128 characters. You should limit the database name to 123 characters when no logical filename is supplied, since SQL Server will create a logical filename by appending a suffix to the database name.

{ON | ADD} *file_definition* [,...]

Defines the disk file(s) that store the data components of the database for *CREATE DATABASE* or adds disk file(s) for *ALTER DATABASE*. *ON* is required for *CREATE DATABASE* only if you wish to provide one or more file definitions, where the syntax for *file_definition* is:

```
{[PRIMARY] ( [NEW][NAME = file_name] [, FILENAME = 'os_file_name']  
[, SIZE = int[KB | MD | GB | TB] ] [, MAXSIZE = { int |  
UNLIMITED } ] [, FILEGROWTH = int] ) } [,...]
```

where:

PRIMARY

Defines the *file_definition* as the primary file. Only one is allowed per database. (If you don't define a primary file, SQL Server defaults primary status to the file that it autocreates, in the absence of any user-defined file, or to the first file that you define.) The primary file, or groups of files (also called filegroups), contains the logical start of the database, all the database system tables, and all other objects not contained in user filegroups.

[NEW]NAME = *file_name*

Defines the logical name of the file defined by the *file_definition* for *CREATE DATABASE*. Use *NEWNAME* for *ALTER DATABASE* to define a new logical name for the file. In either case, the logical name must be unique within the database. This clause is optional when using *FOR ATTACH*.

FILENAME = '*os_file_name*'

Defines the operating system path and filename for the file defined by *file_definition*. The file must be in a non-compressed directory on the filesystem. For raw partitions, specify only the drive letter of the raw partition.

SIZE = *int*[KB | MB | GB | TB]

Defines the size of the file defined by the *file_definition*. This clause is optional, but defaults to the file size for the primary file of the model database, which is usually very small. Logfiles and secondary datafiles default to 1MB in size. The value of *int* defaults to megabytes; however, you can explicitly define the size of the file using suffixes for kilobyte (KB), megabyte (MB), gigabyte (GB), and terabyte (TB). Size cannot be smaller than 512KB or the size of the primary file of the model database.

MAXSIZE = { *int* | UNLIMITED }

Defines the maximum size the file may grow to. Suffixes, as described under the entry for *SIZE*, are allowed. The default, *UNLIMITED*, allows the file to grow until all available disk space is consumed. Not required for files on raw partitions.

FILEGROWTH = *int*

Defines the growth increment for the file each time it grows. Suffixes, as described under the entry for *SIZE*, are allowed. You may also use the percentage (%) suffix to indicate that the file should grow by the percentage of the total disk space currently consumed. If you omit the *FILEGROWTH* clause, the file will grow in 10% increments, but never less than 64KB. Not required for files on raw partitions.

[ADD] LOG {ON | FILE} *file_definition*

Defines the disk file(s) that store the log component of the database for *CREATE DATABASE* or adds disk file(s) for *ALTER DATABASE*. You can provide one or more *file_definitions* for the transaction logs in a comma-delimited list. Refer to the earlier section under the keyword *ON* for the full syntax of *file_definition*.

REMOVE FILE *file_name*

Removes a file from the database and deletes the physical file. The file must be emptied of all content first.

[ADD] FILEGROUP *filegroup_name file_definition* [,...]

Defines any user filegroups and their file definitions used by the database. All databases have at least one primary filegroup (though many databases only use the primary filegroup that comes with SQL Server by default). Adding filegroups and then moving files to the filegroups can allow you greater control over disk I/O. (We recommend that you do not add filegroups without careful analysis and testing.)

REMOVE FILEGROUP *filegroup_name*

Removes a filegroup from the database and deletes all the files in the filegroup. The files and the filegroup must be empty first.

MODIFY FILE *file_definition*

Changes the definition of a file. This clause is very similar to the {*ADD* | *ON*} *FILE* clause. For example: *MODIFY FILE* (*NAME* = *file_name*, *NEWNAME* = *new_file_name*, *SIZE*=...).

MODIFY NAME = *new_database_name*

Changes the database name from its current name to *new_database_name*.

MODIFY FILEGROUP *filegroup_name* {*filegroup_property* | NAME = *new_filegroup_name* }

Used with *ALTER DATABASE*, this clause has two forms. The first form allows you to change a filegroup name, as in *MODIFY FILEGROUP filegroup_name NAME = new_filegroup_name*. The second form allows you to specify a *filegroup_property* for the filegroup, which must be one of the following:

READONLY

Sets the filegroup to read-only and disallows updates to all objects within the filegroup. *READONLY* can only be enabled by users with exclusive database access and cannot be applied to the primary filegroup.

READWRITE

Disables the *READONLY* property and allows updates to objects within the filegroup. *READWRITE* can only be enabled by users with exclusive database access.

DEFAULT

Sets the filegroup as the default filegroup for the database. All new tables and indexes are assigned to the default filegroup unless explicitly assigned elsewhere. Only one default filegroup is allowed per database. (By default, the *CREATE DATABASE* statement sets the primary filegroup as the default filegroup.)

SET {*state_option* | *cursor_option* | *auto_option* | *sql_option* | *recovery_option*} [,...]

Controls a wide variety of behaviors for the database. These are discussed in the rules and information later in this section.

WITH *termination_option*

Used after the *SET* clause, *WITH* sets the rollback behavior for incomplete transactions whenever the database is in transition. When omitted, transactions must commit or rollback on their own with the database state changes. There are two *termination_option* settings:

ROLLBACK AFTER *int* [SECONDS] | ROLLBACK IMMEDIATE

Causes the database to roll back in *int* number of seconds, or immediately. *SECONDS* is a noise word and does not change the behavior of the *ROLLBACK AFTER* clause.

NO_WAIT

Causes database state or option changes to fail if a change cannot be completed immediately, without waiting for the current transaction to independently commit or roll back.

COLLATE *collation_name*

Defines or alters the default collation used by the database. *Collation_name* can be either a SQL Server collation name or a Windows collation name. By default, all new databases receive the collation of the SQL Server instance. (You can execute the query *SELECT * FROM ::fn_helpcollations()* to see all the collation names available.) To change the collation of a database, you must be the only user in the database, no schema bound objects that depend on the current collation may exist in the database, and the collation change must not result in the duplication of any object names in the database.

FOR LOAD | FOR ATTACH

Places the database in special startup mode. *FOR LOAD* is supported only for backward compatibility with versions earlier than 7.0 and sets the database to a status of loading and "dbo use only." *FOR ATTACH* creates the database from a set of pre-existing operating system files, almost always database files created previously. Because of this, the new database must have the same code page and sort order as the previous database. You only need the *file_definition* of the first primary file or those files that have a different path from the last time the database was attached. In general, you should use the *sp_attach_db* system stored procedure over the *CREATE DATABASE FOR ATTACH* statement unless you need to specify more than 16 *file_definitions*.

The *CREATE DATABASE* command should be issued from the *master* system database. You can, in fact, issue the command *CREATE DATABASE database_name*, with no other clauses, to get a very small, default database.

SQL Server uses *files*, formerly called *devices*, to act as a repository for databases. Files are grouped into one or more filegroups, with at least a *PRIMARY* filegroup assigned to each database. A file is a predefined block of space created on the disk structure. A database may be stored on one or more files or filegroups. SQL Server also allows the transaction log to be placed in a separate location from the database using the *LOG ON* clause. These functions allow sophisticated file planning for optimal control of disk I/O. For example, we can create a database called **sales_report** with a data and transaction logfile:

```
USE master
```

```
GO
```

```
CREATE DATABASE sales_report
```

```
ON
```

```
( NAME = sales_rpt_data, FILENAME =
```

```
'c:\mssql\data\salerptdata.mdf',
```

```
SIZE = 100, MAXSIZE = 500, FILEGROWTH = 25 )
```

```
LOG ON
```

```
( NAME = 'sales_rpt_log',
```

```
FILENAME = 'c:\mssql\log\salesrptlog.ldf',
```

```
SIZE = 25MB, MAXSIZE = 50MB,
```

```
FILEGROWTH = 5MB )
```

```
GO
```

When a database is created, all objects in the model database are copied into the new database. All of the empty space within the file or files defined for the database is then initialized (i.e., emptied out), which means that creating a new and very large database can take a while, especially on a slow disk.

A database always has at least a primary datafile and a transaction logfile, but may also have secondary files for both the data and log components of the database. SQL Server uses default filename extensions: *.mdf* for primary datafiles, *.ndf* for secondary files, and *.ldf* for transaction logfiles. The following example creates a database called **sales_archive**

with several very large files that are grouped into a couple of filegroups:

```
USE master
GO
CREATE DATABASE sales_archive
ON
PRIMARY (NAME = sales_arch1, FILENAME = 'c:\mssql\data\archdata1.mdf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sales_arch2,
        FILENAME = 'c:\mssql\data\archdata2.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sales_arch3,
        FILENAME = 'c:\mssql\data\archdat3.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB)
FILEGROUP sale_rpt_grp1
        (NAME = sale_rpt_grp1_1_data,
        FILENAME = 'c:\mssql\data\SG1Fi1dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sale_rpt_grp1_1_data,
        FILENAME = 'c:\mssql\data\SG1Fi2dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
FILEGROUP sale_rpt_grp2
        (NAME = sale_rpt_grp2_1_data, FILENAME = 'c:\mssql\data\SRG21dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sale_rpt_grp2_2_data, FILENAME = 'c:\mssql\data\SRG22dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
LOG ON
        (NAME = sales_archlog1,
        FILENAME = 'd:\mssql\log\archlog1.ldf',
        SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%),
        (NAME = sales_archlog2,
        FILENAME = 'd:\mssql\log\archlog2.ldf',
        SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%)
GO
```

The *FOR ATTACH* clause is commonly used for situations like a salesperson traveling with a database on a CD. This clause tells SQL Server that the database is attached from an existing operating-system file structure, such as a DVD-ROM, CD-ROM, or portable hard drive. When using *FOR ATTACH*, the new database inherits all the objects and data of the parent database, not the model database.

The following examples show how to change the name of a database, file, or filegroup:

```
-- Rename a database

ALTER DATABASE sales_archive MODIFY NAME = sales_history

GO

-- Rename a file

ALTER DATABASE sales_archive MODIFY FILE

NAME = sales_arch1,

NEWNAME = sales_hist1

GO

-- Rename a filegroup

ALTER DATABASE sales_archive MODIFY FILEGROUP

sale_rpt_grp1

NAME = sales_hist_grp1

GO
```

There may be times when you want to add new free space to a database, especially if you have not enabled it to auto-grow:

```
USE master

GO

ALTER DATABASE sales_report ADD FILE

( NAME = sales_rpt_added01, FILENAME = 'c:\mssql\data\salerptadded01.mdf',

  SIZE = 50MB, MAXSIZE = 250MB, FILEGROWTH = 25MB )

GO
```

When you alter a database, you can set many behavior options on the database. State options (shown as *state_option* in the earlier syntax diagram) control how users access the database. Following is a list of valid state options:

SINGLE_USER | *RESTRICTED_USER* | *MULTI_USER*

Sets the number and type of users with access to the database. *SINGLE_USER* mode allows only one user to access the database at a time. *RESTRICTED_USER* mode allows access only members of the system roles *db_owner*, *dbcreator*, or *sysadmin*. *MULTI_USER*, the default, allows concurrent database access from all users who have permission.

OFFLINE | *ONLINE*

Sets the database to offline (unavailable) or online (available).

READ_ONLY | *READ_WRITE*

Sets the database to *READ_ONLY* mode, where no modifications are allowed, or to *READ_WRITE* mode, where data modifications are allowed. *READ_ONLY* databases can be very fast for query intensive operations, since almost no locking is needed.

Cursor options control default behavior for cursors in the database. In the *ALTER DATABASE* syntax shown earlier, you can replace *cursor_option* with any of the following:

CURSOR_CLOSE_ON_COMMIT { *ON* | *OFF* }

When set *ON*, any open cursors are closed when a transaction commits or rolls back. When set *OFF*, any open

cursors remain open when transactions are committed and open cursors close when a transaction rolls back unless the cursor is *INSENSITIVE* or *STATIC*. (See [DECLARE CURSOR Command](#) later in this chapter for more details.)

CURSOR_DEFAULT { *LOCAL* | *GLOBAL* }

Sets the default scope of all cursors in the database to either *LOCAL* or *GLOBAL*. (See [DECLARE CURSOR Command](#) later in this chapter for more details.)

In the SET clause, *auto_option* controls the automatic file handling behaviors of the database. The following are valid replacements for *auto_option*:

AUTO_CLOSE { *ON* | *OFF* }

When set *ON*, the database automatically shuts down cleanly and frees all resources when the last user exits. When set *OFF*, the database remains open when the last user exits. The default is *OFF*.

AUTO_CREATE_STATISTICS { *ON* | *OFF* }

When set *ON*, statistics are automatically created when SQL Server notices they are missing during query optimization. When set *OFF*, statistics are not created during optimization. The default is *ON*.

AUTO_SHRINK { *ON* | *OFF* }

When set *ON*, the database files may automatically shrink. The database periodically looks for an opportunity to shrink files, though the time is not always predictable. When set *OFF*, files will shrink only when you explicitly and manually shrink them. The default is *OFF*.

AUTO_UPDATE_STATISTICS { *ON* | *OFF* }

When set *ON*, out-of-date statistics are reassessed during query optimization. When set *OFF*, statistics are reassessed only by explicitly and manually recompiling them using the SQL Server command *UPDATE STATISTICS*.

The *sql_options* clause controls ANSI compatibility of the database. You can use the standalone SQL Server command *SET ANSI_DEFAULTS ON* to enable all the ANSI SQL92 behaviors at one time, rather than using the individual statements below. In the SET clause, you can replace *sql_option* with any of the following:

ANSI_NULL_DEFAULT { *ON* | *OFF* }

When set *ON*, the *CREATE TABLE* statement causes columns with no nullability setting to default to NULL. When set *OFF*, the nullability of a column defaults to NOT NULL. The default is *OFF*.

ANSI_NULLS { *ON* | *OFF* }

When set *ON*, comparisons to null yield *UNKNOWN*. When set *OFF*, comparisons to null yield null if both non-UNICODE values are null. The default is *OFF*.

ANSI_PADDING { *ON* | *OFF* }

When set *ON*, strings are padded to the same length for insert or comparison operations on *VARCHAR* and *VARBINARY* columns. When set *OFF*, strings are not padded. The default is *ON*. (We recommend that you *do not* change this!)

ANSI_WARNINGS { *ON* | *OFF* }

When set *ON*, the database warns when problems like "divide by zero" or "null in aggregates" occur. When set *OFF*, these warnings are not raised. The default is *OFF*.

ARITHABORT { *ON* | *OFF* }

When set *ON*, divide-by-zero and overflow errors cause a query or Transact-SQL batch to terminate and roll back any open transactions. When set *OFF*, a warning is raised but processing continues. The default is *ON*. (We

recommend that you *do not* change this!)

CONCAT_NULL_YIELDS_NULL { *ON* | *OFF* }

When set *ON*, returns a null when a null is concatenated to a string. When set *OFF*, nulls are treated as empty strings when concatenated to a string. The default is *OFF*.

NUMERIC_ROUNDABORT { *ON* | *OFF* }

When set *ON*, an error is raised when a numeric expression loses precision. When set *OFF*, losses of precision result in rounding of the result from the numeric expression. The default is *OFF*.

QUOTED_IDENTIFIER { *ON* | *OFF* }

When set *ON*, double quotation marks identify an object identifier with special characters or one that is a reserved word (e.g., a table named **SELECT**). When set *OFF*, identifiers may not contain special characters or reserved words and all occurrences of double quotation marks signify a literal string value. The default is *OFF*.

RECURSIVE_TRIGGERS { *ON* | *OFF* }

When set *ON*, triggers can fire recursively. That is, the actions taken by one trigger may cause another trigger to fire, and so on. When set *OFF*, triggers cannot cause other triggers to fire. The default is *OFF*.

Recovery options control the recovery model used by the database. Use any of the following in place of *recovery_option* in the *ALTER DATABASE* syntax:

RECOVERY { *FULL* | *BULK_LOGGED* | *SIMPLE* }

When set to *FULL*, database backups and transaction logs provide full recoverability even for bulk operations like *SELECT...INTO*, *CREATE INDEX*, etc. *FULL* is the default for SQL Server 2000 Standard Edition and Enterprise Edition. *FULL* provides the most recoverability, even from a catastrophic media failure, but uses more space. When set to *BULK_LOGGED*, logging for bulk operations is minimized. Space is saved and fewer I/Os are incurred, but risk of data loss is greater than under *FULL*. When set to *SIMPLE*, the database can only be recovered to the last full or differential backup. *SIMPLE* is the default for SQL Server 2000 Desktop Edition and Personal Edition.

TORN_PAGE_DETECTION { *ON* | *OFF* }

When set *ON*, SQL Server can detect incomplete I/O operations at the disk level by checking each 512-byte sector per 8K database page. (Torn pages are usually detected in recovery.) The default is *ON*.

For example, we may want to change some behavior settings for the **sales_report** database without actually changing the underlying file structure:

```
ALTER DATABASE sales_report SET ONLINE, READ_ONLY,
```

```
AUTO_CREATE_STATISTICS ON
```

```
GO
```

This statement puts the database online and in read-only mode. It also sets the *AUTO_CREATE_STATISTICS* behavior to *ON*.

See Also

```
CREATE SCHEMA  
DROP STATEMENT
```

CREATE/ALTER FUNCTION/PROCEDURE Statements

The *CREATE FUNCTION* and *CREATE PROCEDURE* statements are very similar in syntax and coding (as are the respective *ALTER* statements).

The *CREATE PROCEDURE* statement creates a *stored procedure*, which takes input arguments and performs conditional processing against various objects in the database. According to the ANSI standard, a stored procedure returns no result set (though it may return a value in an *OUTPUT* parameter). For example, you might use a stored procedure to perform all the processes that close an accounting cycle.

The *CREATE FUNCTION* statement creates a *user-defined function* (UDF), which takes input arguments and returns a single value output in the same way as system-supplied functions like *CAST()* or *UPPER()*. These functions, once created, can be called in queries and data-manipulation operations, such as *INSERT*, *UPDATE*, and the *WHERE* clause of *DELETE* statements. Refer to [Chapter 4](#) for descriptions of built-in SQL functions and their individual vendor implementations.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with variations

SQL2003 Syntax

Use the following syntax to create a stored procedure or function:

```
CREATE {PROCEDURE | FUNCTION} object_name
  ( [{[IN | OUT | INOUT] [parameter_name] datatype [AS LOCATOR] [RESULT]}
    [...]] )
  [ RETURNS datatype [AS LOCATOR]
    [CAST FROM datatype [AS LOCATOR] ]
  [LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
  [PARAMETER STYLE {SQL | GENERAL}]
  [SPECIFIC specific_name]
  [DETERMINISTIC | NOT DETERMINISTIC]
  [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
  [RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
  [DYNAMIC RESULT SETS int]
  [STATIC DISPATCH]
  code_block
```

Use the following syntax to alter a pre-existing UDF or stored procedure:

```
ALTER {PROCEDURE | FUNCTION} object_name
  [( {parameter_name datatype } [...])]
  [NAME new_object_name]
```

[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]

[PARAMETER STYLE {SQL | GENERAL}]

[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]

[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]

[DYNAMIC RESULT SETS *int*]

[CASCADE | RESTRICT]

Keywords

CREATE {PROCEDURE | FUNCTION} [AS LOCATOR] [RESULT] *object_name*

Creates a new stored procedure or user-defined function with the name *object_name*. A user-defined function returns a value, where a stored procedure, in ANSI SQL, does not.

[(IN | OUT | INOUT) [*parameter_name*] *datatype* [,...])

Declares one or more parameters to be passed into a stored procedure in a comma-delimited list enclosed in parentheses. Parameters used with a procedure may pass a value *IN*, *OUT*, or both in and out via *INOUT*.

The syntax for the parameter declaration is:

[{IN | OUT | INOUT}] *parameter_name1 datatype*,

[{IN | OUT | INOUT}] *parameter_name2 datatype*,[...]

To provide an optional *parameter_name*, make sure the name is unique within the stored procedure. The optional *AS LOCATOR* subclause is used to validate an external routine with a *RETURNS* parameter that is a *BLOB*, *CLOB*, *NCLOB*, *ARRAY*, or user-defined type. If you need to change the datatype of a *RETURNS* parameter on the fly, use the *CAST* clause (refer to the function, *CAST*, in [Chapter 4](#)). For example, *RETURNS VARCHAR(12) CAST FROM DATE*. When used with *ALTER*, this clause adds parameters to a pre-existing stored procedure. Refer to [Chapter 2](#) for details on datatypes.

RETURNS *datatype* [AS LOCATOR] [CAST FROM *datatype* [AS LOCATOR]]

Declares the datatype of the result returned by a function. (This clause is used only in the *CREATE FUNCTION* statement and is not used in stored procedures.) The key purpose of a user-defined function is to return a value.

LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}

Declares the language in which the function is written. Most database platforms do not support all of these languages, and often support several not mentioned, like Java. When omitted, the default is SQL. When used with *ALTER*, this clause changes the existing *LANGUAGE* value to the value that you declare.

PARAMETER STYLE {SQL | GENERAL}

Indicates, for external routines only, whether certain implicit and automatic parameters are passed, explicitly, with the options (*SQL*) or not (*GENERAL*). (The difference between SQL style and GENERAL style is that SQL style passes automatic SQL parameters, such as indicators, while the GENERAL style does not automatically pass parameters.) The default is *PARAMETER STYLE SQL*. When used with *ALTER*, this clause changes the existing *PARAMETER STYLE* value to the value that you declare.

SPECIFIC *specific_name*

Uniquely identifies the function generally used with user-defined types.

DETERMINISTIC | NOT DETERMINISTIC

States the nature of values returned by a function. (This clause is used only in *CREATE* and *ALTER FUNCTION* statement.) *DETERMINISTIC* functions always return the same value when given the same parameter values. *NOT DETERMINISTIC* functions may return variable results when given the same parameter values. For example, *CURRENT_TIME* is not deterministic because it returns a constantly advancing value corresponding to the time.

NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA

Specifies, in conjunction with the *LANGUAGE* setting, the type of SQL contained in the user-defined function. When used with *ALTER*, this clause changes the existing *SQL* style value to the value that you declare.

NO SQL

Indicates that no SQL statements of any type are in the function. It is used with a non-SQL *LANGUAGE* setting such as *LANGUAGE ADA...CONTAINS NO SQL*.

CONTAINS SQL

Indicates that SQL statements other than read or modify statements are in the function. This is the default.

READS SQL DATA

Indicates the function contains *SELECT* or *FETCH* statements.

MODIFIES SQL DATA

Indicates the function contains *INSERT*, *UPDATE*, or *DELETE* statements.

RETURN NULL ON NULL INPUT | CALL ON NULL INPUT

These options are for use with a host *LANGUAGE* that cannot support nulls. The *RETURNS NULL ON NULL INPUT* setting causes the function to immediately return a null value if it is passed a null value. The *CALL ON NULL INPUT* setting causes the function to handle nulls according to standard rules, for example returning UNKNOWN when a comparison of two null values occurs. (This clause is used in the *CREATE* and *ALTER PROCEDURE* and *FUNCTION* statements.) When used with *ALTER*, this clause changes the existing NULL style value to the value that you declare.

DYNAMIC RESULT SETS int

Declares that a certain number of cursors, *int*, can be opened by the stored procedure and that those cursors are visible after returning from the procedure. When omitted, the default is *DYNAMIC RESULT SETS 0*. (This clause is not used in *CREATE FUNCTION* statements.) When used with *ALTER*, this clause changes the existing *DYNAMIC RESULT SETS* value to the value that you declare.

STATIC DISPATCH

Returns the static values of a user-defined type or *ARRAY* datatype. Required for non-SQL functions that contain parameters that use user-defined types or *ARRAYS*. (This clause is not used in *CREATE PROCEDURE* statements.) This clause must be the last clause in the function or procedure declaration before the *code_block*.

code_block

Declares the procedural statements that handle all processing within the user-defined function or stored procedure. This is the most important, and usually largest, part of a function or procedure. We assume you're interested in a user-defined function of *LANGUAGE SQL*, so note that the *code_block* may not contain SQL-Schema statements, SQL-Transaction statements, nor SQL-Connection statements. While we assume that you're interested in SQL-based UDFs and stored procedures (this is a SQL book after all), you can declare that the code block is derived externally. The syntax for external *code_blocks* is:

EXTERNAL [*NAME external_routine_name*] [*PARAMETER STYLE*

{*SQL | GENERAL*}] [*TRANSFORM GROUP group_name*]

where:

EXTERNAL [NAME *external_routine_name*]

Defines an external routine and assigns a name to it. When omitted, the unqualified routine name is used.

PARAMETER STYLE [SQL | GENERAL]

Same as for *CREATE PROCEDURE*.

TRANSFORM GROUP *group_name*

Transforms values between user-defined types and host variables in a user-defined function or a stored procedure. When omitted, the default is *TRANSFORM GROUP DEFAULT*.

NAME *new_object_name*

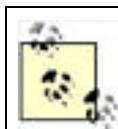
Declares the new name to use for a previously defined UDF or stored procedure. This clause is only used with *ALTER FUNCTION* and *ALTER PROCEDURE* statements.

CASCADE | RESTRICT

Allows you to cause changes to *CASCADE* down to all dependent UDFs or stored procedures, or to *RESTRICT* a change from happening if there are dependent objects. We strongly recommend that you *do not* issue an *ALTER* statement against UDFs or stored procedures that have dependent objects. This clause is only used with *ALTER FUNCTION* and *ALTER PROCEDURE* statements.

Rules at a Glance

With a user-defined function (UDF), you declare the input arguments and the single output argument that the function passes back out. You can then call the user-defined function just as you would any other function, for example, in *SELECT* statements, *INSERT* statements, or the *WHERE* clause of *DELETE* statements.



User-defined functions and stored procedures, when referred to generically, are called *routines*.

With a stored procedure, you declare the input arguments that go into the stored procedure and the output arguments that come out from it. You invoke a stored procedure using the *CALL* statement.

The content of the *code_block* must conform to the rules of whatever procedural language is supported by the database platform. Some platforms do not have their own internal procedural language, requiring you to use *EXTERNAL code_block* constructs.

For example, you might want build a user-defined function on Microsoft SQL Server that returns the first and last name of a person as a single string:

```
CREATE FUNCTION formatted_name (@fname VARCHAR(30),
```

```
@lname VARCHAR(30) )
```

```
RETURNS VARCHAR(60)
```

```
AS
```

```
BEGIN
```

```
    DECLARE @full_name VARCHAR(60)
```



```
SET @full_name = @fname + ' ' + @lname  
RETURN @full_name  
END
```

You could then use this user-defined function as any other function:

```
SELECT formatted_name(au_fname, au_lname) AS name, au_id AS id  
FROM authors
```

The *SELECT* statement in this example will return a result set with the two columns, **name** and **id**.



The ANSI statement *CREATE METHOD* is supported only by DB2 at this time. A method is essentially a special purpose user-defined function and follows the same syntax.

Stored procedures behave similarly. In the following small example, this Microsoft SQL Server stored procedure generates a unique 22-digit value (based on elements of the system date and time) and returns it to the calling process:

```
-- A Microsoft SQL Server stored procedure  
CREATE PROCEDURE get_next_nbr  
    @next_nbr CHAR(22) OUTPUT  
AS  
BEGIN  
    DECLARE @random_nbr INT  
    SELECT @random_nbr = RAND( ) * 1000000  
    SELECT @next_nbr =  
        RIGHT('000000' + CAST(ROUND(RAND(@random_nbr)*1000000,0))AS  
            CHAR(6), 6) +  
        RIGHT('0000' + CAST(DATEPART (yy, GETDATE( ) ) AS CHAR(4)), 2) +  
        RIGHT('000' + CAST(DATEPART (dy, GETDATE( ) ) AS CHAR(3)), 3) +  
        RIGHT('00' + CAST(DATEPART (hh, GETDATE( ) ) AS CHAR(2)), 2) +  
        RIGHT('00' + CAST(DATEPART (mi, GETDATE( ) ) AS CHAR(2)), 2) +  
        RIGHT('00' + CAST(DATEPART (ss, GETDATE( ) ) AS CHAR(2)), 2) +  
        RIGHT('000' + CAST(DATEPART (ms, GETDATE( ) ) AS CHAR(3)), 3)  
END  
GO
```

In this next, and final, ANSI SQL2003 example, we change the name of an existing stored procedure:

```
ALTER PROCEDURE get_next_nbr  
NAME 'get_next_ID'  
RESTRICT;
```

Programming Tips and Gotchas

A key advantage of a stored procedure or function is the fact that it is *precompiled*, meaning that once created its query plans are already stored in the database. Precompiled routines can often (though not always) be cached in database memory to provide an additional boost in performance. A stored procedure or user-defined function can perform many statements with a single communication to the server, thus reducing network traffic.

Implementations of user-defined functions and stored procedures vary widely by platform. Some database platforms do not support internal *code_block* content. On these platforms, you can only write an external *code_block*. Be sure to read about the variations and capabilities on each platform.

If you execute an *ALTER PROCEDURE* or *FUNCTION* statement, dependent objects may become invalid after a change to an object they depend on. Be careful to check all dependencies when altering UDFs or stored procedures on which other UDFs or stored procedures may depend.

DB2

DB2's implementations of these commands are very close to the ANSI standard. Although the syntax diagrams are long, much of the usage is identical to the ANSI standard. Consequently, the keyword explanation section that follows the syntax diagram is relatively short, since many of the clauses are explained in the SQL standard.

DB2 supports several classifications of functions. Each kind of UDF has its own set of allowable syntax, though much of it is shared across the various kinds of UDF. DB2 allows external scalar functions, external table functions, OLE-DB external table functions, sourced (or template) functions, and SQL functions. DB2 supports two kinds of stored procedures: external and SQL.

The following syntax is the full syntax to create any sort of stored procedure or a UDF in DB2. It is grouped together here to give you a concise list of clause definitions. However, refer to the sections below for greater detail on the specific allowable syntax for each type of UDF and stored procedure. If you prefer, for example, to see only the allowable syntax for a SQL user-defined function, then skip to the section below. The total available syntax for UDFs and stored procedures follows:

```
CREATE {PROCEDURE | FUNCTION} object_name

    [( { [IN | OUT] INOUT } [parameter_name] datatype } [...])]

RETURNS {datatype | {ROW | TABLE} column_list} [AS LOCATOR]

    [CAST FROM datatype [AS LOCATOR] ]

[LANGUAGE {C | JAVA | COBOL | OLE | SQL}]

[PARAMETER STYLE {SQL | DB2GENERAL | GENERAL | GENERAL WITH NULLS | JAVA}]

[SPECIFIC specific_name]

[EXTERNAL NAME external_routine_name]

[DETERMINISTIC | NOT DETERMINISTIC]

[ [NO] EXTERNAL ACTION]

[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]

[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]

[DYNAMIC RESULT SETS int]

[STATIC DISPATCH]

[SOURCE object_name | AS TEMPLATE ]

[INHERIT SPECIAL REGISTERS]

[[NOT] FENCED] [[NOT] THREADSAFE]

[PROGRAM TYPE {SUB | MAIN}]

[ [NO] SCRATCHPAD int] [ [NO] FINAL CALL]
```

[{ALLOW | DISALLOW} PARALLEL]

[[NO] DBINFO]

[TRANSFORM GROUP *group_name*]

[CARDINALITY *int*]

[PREDICATES (*predicate_definition*)]

code_block

In addition, use the following syntax to alter a pre-existing UDF or stored procedure, and assign it the characteristics that you declare:

ALTER {PROCEDURE | FUNCTION} *object_name*

[EXTERNAL NAME *external_routine_name*]

[[NOT] FENCED] [[NOT] THREADSAFE]

Following are the parameter descriptions for DB2:

RETURNS {*datatype* | {ROW | TABLE} *column_list*}

Allows you to return a single value of *datatype*, a ROW of values, or a TABLE of values where *column_list* uses the syntax *column_name1 datatype1, column_name2 datatype2 [...]*.

LANGUAGE {C | JAVA | COBOL | OLE | SQL}

Same as the ANSI standard, though DB2 supports other languages than in the ANSI standard. When you declare a LANGUAGE setting, other options are restricted or made available.

PARAMETER STYLE {SQL | DB2GENERAL | GENERAL | GENERAL WITH NULLS | JAVA}

Similar to the ANSI SQL standard but with a richer variety of choices. PARAMETER STYLE is only used with external stored procedures or user-defined functions. The various styles determine how parameters are passed, where:

SQL

Defines that when parameters are passed with the CALL statement, a null indicator for each parameter and a qualified and specific name for the stored procedure or function are also passed. In addition, the SQLSTATE and SQL diagnostic string are passed back to DB2.

DB2GENERAL

Available only with the LANGUAGE JAVA clause, the DB2GENERAL clause specifies that the parameters are passed using the convention defined for Java.

GENERAL

Passes parameters directly from the CALL statement to the stored procedure without using a SQLDA structure. Nulls are not passed. Only available with LANGUAGE C or COBOL.

GENERAL WITH NULLS

Same as GENERAL, but also passes null indicators.

JAVA

Passes parameters using the Java and SQLJ Routines specifications. For example, *INOUT* and *OUT* parameters are passed as a single entry array. Available only with the *LANGUAGE JAVA* clause. This clause is mutually exclusive of the *DBINFO* clause and *PROGRAM TYPE* clause.

EXTERNAL NAME [*external_routine_name*]

Registers a new function or procedure using a linked and interfaced external program. When *external_routine_name* is omitted, *object_name* is assumed to be the name of the external program. The method of linking depends on the *LANGUAGE* setting.

NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA

Same as the ANSI standard, though certain options are not usable for certain *LANGUAGE* settings.

INHERIT SPECIAL REGISTERS

Declares that updateable special registers receive their initial value from the invoking process. The invoking process could be a UDF, a stored procedure, trigger, view, or user issuing a *CALL* statement.

[[NOT] FENCED] [{THREADSAFE | NOT THREADSAFE}]

Specifies whether (or *NOT*) the external stored procedure or user-defined function is "safe" to run in DB2's manager operating environment process or address space. This is used only for external stored procedures. *NOT FENCED* procedures tend to be faster, but open the database up to potential risks.

[[NOT] THREADSAFE]

Declares whether the method is safe to run in the same process as other routines (*THREADSAFE*) or not (*NOT THREADSAFE*). When declared *THREADSAFE*, a procedure is considered safe to run in the same process as other UDFs and stored procedures. *NOT THREADSAFE* can only be used with *FENCED* stored procedures.

PROGRAM TYPE {SUB | MAIN}

Tells whether the external stored procedure should expect parameters in the style of a *SUB* routine or a *MAIN* routine. *SUB*, the default, declares that the parameters are expected as separate arguments. *MAIN* declares that the parameters are expected as an argument counter and argument vector. *MAIN* also requires that the external stored procedure is named "main" and that it is built as a shared library, not as a standalone executable. *PROGRAM TYPE MAIN* is only valid with *LANGUAGE C* or *COBOL* and *PARAMETER STYLE GENERAL*, *GENERAL WITH NULLS*, *SQL*, or *DB2SQL*.

[NO] SCRATCHPAD *int*

Provides a scratch pad for any external function or procedure. A *SCRATCHPAD* provides a way for a UDF to save its state from one call to the next. The default is *NO SCRATCHPAD*. If an *int* is specified, it represents the size of the scratch pad in bytes (defaulting to 100 bytes). One scratch pad is initialized per reference to an external routine.

[NO] FINAL CALL

FINAL CALL causes the routine to invoke two extra times. The first invocation occurs before data is first loaded so the routine can perform any one-time initialization needed. The second invocation occurs last and allows the routine to do any cleanup needed. Table functions are always a *FINAL CALL*. This clause is incompatible with *PARAMETER STYLE JAVA*. The clause *NO FINAL CALL* prevents a final call from happening.

{ALLOW | DISALLOW} PARALLEL

Allows the routine to be split in parallel across multiple CPUs (a.k.a. parallelized) during plan optimization on an SMP or MPP server, with *ALLOW PARALLEL*. Otherwise, DB2 enforces the serial processing of a routine on a single CPU, regardless of how many CPUs exist on the server, with *DISALLOW PARALLEL*. *ALLOW PARALLEL* is the default, unless the routine contains any of these clauses: *NOT DETERMINISTIC*, *EXTERNAL ACTION*, *SCRATCHPAD*, or *FINAL CALL*.

[NO] DBINFO

Tells whether the external stored procedure receives an additional argument (*DBINFO*) or not (*NO DBINFO*, the default) when invoked. The *DBINFO* argument tells the database name, application ID, application authorization ID, code page, database release version, and platform. *DBINFO* is mutually exclusive of *LANGAUGE OLD* and *PARAMETER STYLE JAVA* and *DB2GENERAL*.

TRANSFORM GROUP *group_name*

Declares the transform group, *group_name*, if the routine uses a user-defined structured type for either a parameter or return datatype.

CARDINALITY *int*

PREDICATES (*predicate_definition*)

Used with *CREATE FUNCTION* to create a user-defined predicate. This option is used when defining user-defined functions for use with DB2's index extensions. Index extensions are quite complex and not used by the average DB2 user. Consequently, you should refer to DB2's documentation on index extensions for information on this clause. In addition, *DETERMINISTIC* with *NO EXTERNAL ACTION* clauses are required to use *PREDICATES*.

DB2 accommodates several kinds of user-defined functions: external scalar, external table, OLE-DB, source/template, and SQL, and two kinds of stored procedures: external and SQL. Each of these types allows a mix of only some of the clauses shown above.

DB2 external scalar user-defined functions

This variation of the *CREATE FUNCTION* statement is used to register a user-defined external scalar function with an application server. (Remember that *scalar functions* return a single value each time they are invoked and are valid wherever SQL expressions are valid.) An external scalar UDF allows the following clauses:

CREATE FUNCTION object_name (parameter_name datatype

[,...]) [AS LOCATOR]

RETURNS datatype [AS LOCATOR] [CAST FROM datatype

[AS LOCATOR]]

[LANGUAGE {C | JAVA | OLE}] [PARAMETER STYLE

{SQL | DB2GENERAL | JAVA}]

[SPECIFIC specific_name]

[EXTERNAL NAME external_routine_name]

[DETERMINISTIC | NOT DETERMINISTIC] [STATIC DISPATCH]

[[NO] EXTERNAL ACTION]

[NO SQL | CONTAINS SQL | READS SQL DATA]

[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]

[INHERIT SPECIAL REGISTERS]

[[NOT] FENCED [{THREADSAFE | NOT THREADSAFE}]]

[[NO] SCRATCHPAD int] [[NO] FINAL CALL]

[{ALLOW | DISALLOW} PARALLEL]

[[NO] DBINFO] [TRANSFORM GROUP group_name]

[PREDICATES (predicate_definition)]

code_block

The following code is an example of an external scalar UDF, called **report_reader.center**, that defines the center of a circle. The statement explicitly declares all option clauses, even those that have identical defaults to the declared values:

```
CREATE FUNCTION report_reader.center (FLOAT, FLOAT, FLOAT)
RETURNS DECIMAL(8,4) CAST FROM FLOAT
SPECIFIC center04
EXTERNAL NAME 'center!focalpt'
LANGUAGE C PARAMETER STYLE SQL DETERMINISTIC FENCED NOT NULL CALL
NO SQL NO EXTERNAL ACTION SCRATCHPAD NO FINAL CALL
DISALLOW PARALLEL
```

Note that the **report_reader.center** UDF uses a scratch pad and can accumulate and act on results stored there

DB2 external table user-defined functions

This variation of the *CREATE FUNCTION* statement is used to register a user-defined external table function with an application server. (A *table function* can be used in the *FROM* clause of a *SELECT* statement, returning a table to the *SELECT* one row at a time.) An external table UDF allows the following clauses:

```
CREATE FUNCTION object_name (parameter_name datatype }
    [AS LOCATOR] [...])
RETURNS TABLE datatype [AS LOCATOR] [...] ]
[LANGUAGE {C | JAVA | OLE} ] [PARAMETER STYLE
    {SQL | DB2GENERAL}]
[SPECIFIC specific_name] [EXTERNAL NAME external_routine_name]
[DETERMINISTIC | NOT DETERMINISTIC] [ [NO] EXTERNAL ACTION]
[NO SQL | CONTAINS SQL | READS SQL DATA]
[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]
[INHERIT SPECIAL REGISTERS] [ [NOT] FEDERATED]
[ [NOT] FENCED [{THREADESAFE | NOT THREADESAFE}] ]
[ [NO] SCRATCHPAD inf] [ [NO] FINAL CALL]
    [DISALLOW PARALLEL]
[ [NO] DBINFO] [TRANSFORM GROUP group_name]
[CARDINALITY inf]
code_block
```

Following is an example external table UDF that retrieves message header information and some message text from Microsoft Exchange:

```
CREATE FUNCTION MAIL( )
RETURNS TABLE (time_rcvd DATE, subjct VARCHAR(15),
    lngth INTEGER, txt VARCHAR(30))
EXTERNAL NAME 'mail.header!list'
LANGUAGE OLE
PARAMETER STYLE SQL
NOT DETERMINISTIC FENCED CALLED ON NULL INPUT
```

SCRATCHPAD FINAL CALL

NO SQL

EXTERNAL ACTION DISALLOW PARALLEL

Note that the external function, **mail.header!list**, should already exist.

DB2 OLE-DB external table user-defined functions

This variation of the *CREATE FUNCTION* statement registers a user-defined OLE-DB external table function to access data from an OLE-DB provider. The table function may then be used in the *FROM* clause of a *SELECT* statement. This is useful to link to non-DB2 sources of relational data. An OLE-DB external table UDF allows the following clauses:

```
CREATE FUNCTION object_name (parameter_name datatype) [,...]
```

```
RETURNS TABLE datatype [,...] ] [SPECIFIC specific_name]
```

```
[LANGUAGE OLEDB] [EXTERNAL NAME external_routine_name]
```

```
[DETERMINISTIC | NOT DETERMINISTIC] [ [NO] EXTERNAL ACTION]
```

```
[STATIC DISPATCH] [CALLED ON NULL INPUT |
```

```
RETURNS NULL ON NULL INPUT]
```

```
[CARDINALITY inf]
```

```
code_block
```

The following example registers an OLE-DB table function that retrieves **order** information from the Microsoft Access database called **Northwind**:

```
CREATE FUNCTION orders ( )
```

```
RETURNS TABLE (orderid INTEGER, customerid CHAR(5), employeeid INTEGER,
```

```
orderdate TIMESTAMP, requireddate TIMESTAMP, shippeddate TIMESTAMP,
```

```
shipvia INTEGER, freight DEC(19,4))
```

```
LANGUAGE OLEDB
```

```
EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
```

```
Data Source=c:\sql\lib\samples\oledb\nwind.mdb';
```

Note that the connection string is defined in the external name of the function.

DB2 sourced user-defined functions

This variation of the *CREATE FUNCTION* statement registers a user-defined function that is based on another existing scalar or column function. This statement can also be used to register a function template with an application server that acts as a federated server. (A *function template* is a partial function that has no executable code, but instead uses it to map to a data source function. Once the mapping is created, the user specifies the function template in queries to the federated server. The federated server will invoke the data source function and return values that correspond to those in the *RETURNS* portion of the template definition.) A source/template UDF allows the following clauses:

```
CREATE FUNCTION object_name (parameter_name datatype [,...])
```

```
RETURNS datatype [,...] ]
```

```
[SOURCE {object_name [input_argument [,...] ] |
```

```
[SPECIFIC specific_name] ]
```

```
[AS TEMPLATE [DETERMINISTIC | NOT DETERMINISTIC]
 [ [NO] EXTERNAL ACTION] ]
```

code_block

For example, the following sourced UDF is based on the **report_reader.center** scalar function created earlier in this section. The following function, called **fn_center**, accepts only integer arguments:

```
CREATE FUNCTION fn_center (INTEGER, INTEGER)
RETURNS FLOAT
SOURCE report_reader.center (INTEGER, FLOAT);
```

In the next example, using a federated server, we can invoke an Oracle UDF, called *statistics* in the schema *scott*, that returns table statistics as a double-precision floating point value:

```
CREATE FUNCTION scott.statistics (DOUBLE, DOUBLE)
RETURNS DOUBLE AS TEMPLATE
DETERMINISTIC NO EXTERNAL ACTION;
```

DB2 SQL user-defined functions

This variation of the *CREATE FUNCTION* statement defines a SQL scalar, table, or row function. This application shows the most common form of SQL UDF's in DB2. (Remember a *scalar function* returns a single value when invoked. A *table function* may be used in the *FROM* clause of a *SELECT* statement and returns a table. A *row function* returns a row and may be used as a transform function.)

A SQL UDF allows the following clauses:

```
CREATE FUNCTION object_name [(parameter_name datatype [...])]
RETURNS {datatype | {ROW | TABLE} column_list}
    [SPECIFIC specific_name]
    [LANGUAGE SQL] [DETERMINISTIC | NOT DETERMINISTIC]
    [ [NO] EXTERNAL ACTION]
    [CONTAINS SQL | READS SQL DATA] [STATIC DISPATCH]
    [CALLED ON NULL INPUT] [PREDICATES (predicate_definition) ]
```

code_block

Here's an example of a SQL user-defined function that returns a concatenated name from separate fields:

```
CREATE FUNCTION prod.cust_name
(in_custno INTEGER )
RETURNS VARCHAR(31)
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
DETERMINISTIC
```



```
RETURN

SELECT (c.firstnme CONCAT ' '
        CONCAT c.midinit CONCAT ' '
        CONCAT c.lastname)
AS name
FROM prod.customer c
WHERE c.custno = 10;
```

DB2 external stored procedure

This variation of the *CREATE PROCEDURE* statement defines an external procedure with a DB2 application server. Usually, external procedures are written in a non-SQL language like C or Java. An external procedure allows the following clauses:

```
CREATE PROCEDURE object_name
  ([ [IN | OUT | INOUT] parameter_name datatype [,...] ] )
  [SPECIFIC specific_name] [DYNAMIC RESULT SETS int]
  [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
  [DETERMINISTIC | NOT DETERMINISTIC] [CALLED ON NULL INPUT]
LANGUAGE {C | JAVA | COBOL | OLE}
  EXTERNAL NAME external_routine_name
  [INHERIT SPECIAL REGISTERS]
PARAMETER STYLE {SQL | DB2GENERAL | DB2SQL | GENERAL |
  GENERAL WITH NULLS | JAVA}
  [ [NOT] FENCED [{THREADSAFE | NOT THREADSAFE}] ]
  [PROGRAM TYPE {SUB | MAIN}]
  [ [NO] DBINFO]
code_block
```

Following is an example of an external procedure. In the example, we define a procedure definition in DB2 for a C program that, when passed an author ID, tells us the total books sold, the total sales in dollars, and the book titles the specified author wrote:

```
CREATE PROCEDURE author_sales
  (IN au_id INTEGER, OUT books_sold INTEGER,
  OUT total_sales DOUBLE, OUT title VARCHAR(50) )
EXTERNAL NAME 'authors!sales'
DYNAMIC RESULT SETS 1 NOT FENCED
LANGUAGE C PARAMETER STYLE GENERAL
```

DB2 SQL stored procedure

This variation of the *CREATE PROCEDURE* statement defines a SQL-based procedure with a DB2 application server. A SQL procedure allows the following clauses:

```
CREATE PROCEDURE object_name [( [IN |
    OUT| INOUT] parameter_name datatype [,...] ) ]
    [SPECIFIC specific_name] [DYNAMIC RESULT SETS int]
    [CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
    [DETERMINISTIC | NOT DETERMINISTIC] [CALLED ON NULL INPUT]
    [INHERIT SPECIAL REGISTERS] [LANGUAGE SQL]
code_block
```

The following SQL stored procedure returns a customer's record based upon a *SOUNDEX* function on their name and state:

```
CREATE PROCEDURE prod.custlookup
(IN in_lastname VARCHAR(15),
 IN in_st CHARACTER(2) )
DYNAMIC RESULT SETS 1
LANGUAGE SQL
NOT DETERMINISTIC
CALLED ON NULL INPUT
MODIFIES SQL DATA
INHERIT SPECIAL REGISTERS
P1: BEGIN
    DECLARE cursor1 CURSOR WITH RETURN FOR
        SELECT *
        FROM prod.customer c
        WHERE SOUNDEX (c.lastname) = SOUNDEX (in_lastname)
        AND c.st = UPPER (in_st);
    OPEN cursor1;
END P1;
```

DB2 has recently implemented new procedural extensions to SQL. You can now define the *code_block* for a DB2 UDF or stored procedure using conditional processing constructs like *IF...THEN* and *WHILE* loops (as shown in the examples above).

RETURNS TABLE is available as a return value for any *LANGUAGE*. The only clauses allowed are *STATIC DISPATCH*, *DETERMINISTIC*, *EXTERNAL ACTION*, *RETURNS NULL ... | CALLED NULL... ON INPUT*, and *CARDINALITY*.

READ SQL DATA and *CONTAINS SQL* are the only SQL-style options allowed when you chose *LANGUAGE SQL*. These clauses disable the *CAST* clause and the *RETURNS NULL ON NULL INPUT* clause. It also makes available (though optional) the *RETURNS {ROW | TABLE}* clause, *FEDERATED* clause, *PREDICATES* clause, and *INHERIT SPECIAL REGISTERS* clause.

ALTER FUNCTION and *PROCEDURE* on DB2 merely change the characteristics of an existing UDF or stored procedure. The two examples shown next make a function and a procedure, respectively, *NOT FENCED*.

```
ALTER FUNCTION mail( ) NOT FENCED;
```

```
ALTER PROCEDURE employee_pay NOT FENCED;
```

MySQL

MySQL does not support the *ALTER FUNCTION* statement, nor does it support the *CREATE PROCEDURE* and *ALTER PROCEDURE* statements.

All user-defined functions created on a MySQL database are, in effect, external function calls:

```
CREATE [AGGREGATE] FUNCTION function_name
RETURNS {STRING | REAL | INTEGER}
SONAME shared_program_library_name ;
```

where:

```
CREATE [AGGREGATE] FUNCTION function_name
```

Creates an external function with a *function_name* of not more than 64 characters. The module will be stored in the *mysql2/udf* directory, with an extension of *.udf*. If *AGGREGATE* is specified, the user-defined function must return an aggregate value like *SUM()* or *AVG()*.

```
RETURNS {STRING | REAL | INTEGER}
```

Returns either *STRING* for character data, *REAL* for floating-point numbers, or *INTEGER* for whole numbers.

```
SONAME shared_program_library_name
```

Names the C/C++ program that performs the processing. This *shared_program* can be compiled and built into the MySQL engine, or it can function as an independent, linked program.

Once implemented, a MySQL UDF may be called just like any built-in function, such as *ABS()* or *SOUNDEX()*.

The implementation of *CREATE FUNCTION* in MySQL is dependent on procedural code in C/C++ under an operating system that supports dynamic loading. The C/C++ program is named in the *shared_program_library_name* option. The function may be compiled either directly into the MySQL server, making the function permanently available, or as a dynamically callable program. For example, the code behind the UDF created in the following statement might be found on a Unix server:

```
CREATE FUNCTION find_radius RETURNS INT SONAME "radius.so";
```

Oracle

Oracle supports *ALTER* and *CREATE* for both *FUNCTION* and *PROCEDURE* object types. (You may also wish to learn about Oracle packages, which can also be used to create UDFs and stored procedures. Check Oracle's documentation.) Oracle's *CREATE PROCEDURE* syntax is as follows:

```
CREATE [OR REPLACE] {FUNCTION | PROCEDURE} [schema.]object_name
[(parameter1 [IN | OUT | IN OUT] [NOCOPY] datatype [,... ] ) ] ]
RETURN datatype
[DETERMINISTIC] [AUTHID {CURRENT_USER | DEFINER}]
[PARALLEL_ENABLE [( PARTITION prtn_name BY {ANY | {HASH | RANGE}
( column [,... ] ) } {ORDER | CLUSTER} BY ( column [,... ] ) ) ] ]
{ [PIPELINED | AGGREGATE] USING [schema.]implementation_type] |
[PIPELINED] {IS | AS} }
{code_block | LANGUAGE {JAVA NAME external_program_name |
```

```
C [NAME external_program_name]  
LIBRARY lib_name [AGENT IN (argument [,...])] [WITH CONTEXT]  
[PARAMETERS ( param [,...] ) ] ;
```

The *ALTER FUNCTION* and *ALTER PROCEDURE* statement shown next is used to recompile invalid UDFs or stored procedures:

```
ALTER {FUNCTION | PROCEDURE} [schema.]object_name  
COMPILE [DEBUG] [compiler_param = value [...]] [REUSE SETTINGS]
```

Following are the parameter descriptions:

CREATE [OR REPLACE] FUNCTION [*schema*.]*object_name*

Creates a new UDF or stored procedure. Use *OR REPLACE* to replace an existing procedure or UDF without first dropping it, and then having to reassign all permissions to it. Certain clauses are only used with user-defined functions, including the *RETURN* clause, the *DETERMINISTIC* clause, and the entirety of the *USING* clause.

[IN | OUT | IN OUT]

Specifies whether a parameter is an input to the function, an output from the function, or both.

NOCOPY

Speeds up performance when an *OUT* or *IN OUT* argument is very large, as with a *VARRAY* or *RECORD* datatype.

AUTHID {*CURRENT_USER* | *DEFINER*}

Forces the routine to run in the permission context of either the current user or the person who owns the function, using *AUTHID CURRENT_USER* or *AUTHID DEFINER*, respectively.

PARALLEL_ENABLE

Enables the routine to be executed by a parallel query operation on an SMP or parallel-processor server. This clause is used only for UDFs. (Don't use session state or package variables because they can't be expected to be shared among parallel execution servers.) Define the behavior of the *PARALLEL_ENABLE* query operation using these subclauses, where:

PARTITION *prtn_name* BY {ANY | {HASH | RANGE} (*column* [,...])}

Defines partitioning of inputs on functions with *REF CURSOR* arguments. This may benefit table functions. *ANY* allows random partitioning. You can restrict partitioning to a specific *RANGE* or *HASH* partition on a comma-delimited *column* list.

{ORDER | CLUSTER} BY (*column* [,...])

Orders or clusters parallel processing based on a comma-delimited *column* list. *ORDER BY* causes the rows to be locally ordered on the parallel execution server according to the *column* list. *CLUSTER BY* restricts the rows on the parallel execution server to the key values identified in the *column* list.

{PIPELINED | AGGREGATE} [USING [*schema*.] *implementation_type*]

PIPELINED iteratively returns the results of a table function, instead of the normal serial return of the *VARRAY* or nested table result set. This clause is only used for UDFs. The clause *PIPELINED USING implementation_type* is for an external UDF that uses a language like C++ or Java. The *AGGEGATE USING implementation_type* clause defines a UDF as an aggregate function (a function that evaluates many rows, but returns a single value).

IS | AS

Oracle treats *IS* and *AS* equally. Use either one to introduce the *code_block*.

code_block

Oracle allows a PL/SQL code block for user-defined functions and stored procedures. Alternately, you may use the *LANGUAGE* clause for stored procedures written in Java or C.

```
LANGUAGE {JAVA NAME external_program_name | C [NAME external_program_name] LIBRARY lib_name [AGENT IN  
(argument) [WITH CONTEXT] [PARAMETERS ( params ) ] }
```

Defines the Java or C implementation of the external program. The parameters and semantics of each declaration are specific to Java and C, not SQL.

```
ALTER {FUNCTION | PROCEDURE} [schema.]object_name
```

Recompiles an invalid standalone stored routine. Use *CREATE ... OR REPLACE* to change the arguments, declarations, or definition of an existing routine.

```
COMPILE [DEBUG] [REUSE SETTINGS]
```

Recompiles the routine. Note that *COMPILE* is required. (You can see compile errors with the SQL*Plus command *SHOW ERRORS*.) The routine is marked valid if no compiler errors are encountered. The following optional subclauses may also be included with the *COMPILE* clause, where:

DEBUG

Generates and stores code used by the PL/SQL debugger.

```
compiler_param = value [...]
```

Specifies a PL/SQL compiler parameter. Allowable parameters include *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. Refer to Oracle's documentation on the PL/SQL compiler for more details.

REUSE SETTINGS

Maintains the existing compiler switch settings and reuses them for recompilation. Normally, Oracle drops and reacquires compiler switch settings.

In Oracle, UDFs and stored procedures are very similar in composition and structure. The primary difference is that a stored procedure cannot return a value to the invoking process, while a function may return a single value to the invoking process.

For example, you can pass in the name of a construction project to the following function to obtain the project's profit:

```
CREATE OR REPLACE FUNCTION project_revenue (project IN varchar2)
```

```
RETURN NUMBER
```

```
AS
```

```
    proj_rev NUMBER(10,2);
```

```
BEGIN
```

```
    SELECT SUM(DECODE(action,'COMPLETED',amount,0)) -
```

```
           SUM(DECODE(action,'STARTED',amount,0)) +
```

```
           SUM(DECODE(action,'PAYMENT',amount,0))
```

```
INTO proj_rev  
  
FROM construction_actions  
  
WHERE project_name = project;  
  
RETURN (proj_rev);  
  
END;
```

In this example, the user-defined function accepts the project name as an argument. Then it processes the project revenue, behind the scenes, by subtracting the starting costs from the completion payment and adding any other payments into the amount. The `RETURN (proj_rev);` line returns the amount to the invoking process.

In Oracle, UDFs *cannot* be used in the following situations:

- In a *CHECK* constraint or *DEFAULT* constraint of a *CREATE TABLE* or *ALTER TABLE* statement.
- In a *SELECT*, *INSERT*, *UPDATE*, or *DELETE*, the UDF cannot, either directly or indirectly (if invoked by another routine):
 - Have an *OUT* or *IN OUT* parameter. (Indirect calls may take *OUT* and *IN OUT* parameters.)
 - Terminate the transaction with *COMMIT*, *ROLLBACK*, *SAVEPOINT*, or a *CREATE*, *ALTER*, or *DROP* statement that implicitly issues a *COMMIT* or *ROLLBACK*.
 - Use session control (*SET ROLE*) or system control (the Oracle-specific statement *ALTER SYSTEM*) statements.
 - Write to a database (when a component of a *SELECT* statement or a parallelized *INSERT*, *UPDATE*, or *DELETE* statement).
 - Write to the same table that is modified by the statement that calls the UDF.

Note that when you recompile a routine with the *ALTER* statement, it is marked valid if no compiler errors are encountered. (It is marked invalid if any errors are encountered.) Perhaps more importantly, any objects that depend upon the recompiled routine are marked invalid, regardless of whether an error occurs. You can either recompile those dependent objects yourself, or you can allow Oracle to take some additional time to recompile them at runtime.

By way of example, the following recompiles the **project_revenue** function and maintains any compiler information for the PL/SQL debugger:

```
ALTER FUNCTION project_revenue COMPILE DEBUG;
```

PostgreSQL

PostgreSQL supports the *CREATE FUNCTION* statement, but not *CREATE PROCEDURE*, *ALTER FUNCTION*, or *ALTER PROCEDURE*. Since PostgreSQL does not have an internal procedural language, user-defined functions are usually external functions, though they may be simple SQL queries. The syntax to use to create a function is:

```
CREATE [OR REPLACE] FUNCTION function_name  
  
    ( [ parameter [, ...] ] )  
  
RETURNS datatype  
  
AS { code_block | object_file, link_symbol }  
  
LANGUAGE { C | SQL | PLPGSQL | PLTCL | PLTCLU | PLPERL |  
  
    internal }  
  
[WITH { [ISCACHEABLE] [,] [ISSTRICT] } ]
```

where:

CREATE [OR REPLACE] *function_name*

Creates a new function of the name you provide, or replaces an existing function. *OR REPLACE* does not enable you to change the name, input parameters, or output results of an existing function. You must drop and recreate a function to change any of those settings.

RETURNS *datatype*

Specifies the type of data required by the function.

AS {*code_block* | *object_file*, *link_symbol*}

Defines the composition of the user-defined function. The *code_block* can be a string defining the function (dependent on the *LANGUAGE* setting), such as an internal function name, the path and name of an object file, a SQL query, or the text of a procedural language. The definition also can be an *object file* and *link symbol* for a C-language function.

LANGUAGE {*C* | *SQL* | *PLPGSQL* | *PLTCL* | *PLTCLU* | *PLPERL* | *internal*}

Defines a call to an external program or an internal SQL routine. Since all of the language options, except *SQL*, are programs compiled in other languages, they are beyond the scope of this book. However, the *LANGUAGE SQL* clause should be used when writing *SQL* user-defined functions.



Add a new language not installed by default on PostgreSQL by using the *CREATE LANGUAGE* statement.

[WITH {[*ISCACHABLE*] [,] [*ISSTRICT*]}]

Optimizes PostgreSQL performance by indicating that the function always returns the same value when provided with the same parameter values. *WITH ISCACHABLE* is similar to the ANSI setting *DETERMINISTIC*, except that this setting then allows the optimizer to pre-evaluate the call of the function. *WITH ISSTRICT* is similar to the ANSI setting *RETURNS NULL ON NULL INPUT*. When omitted, the default behavior is similar to *CALLED ON NULL INPUT*. Note that you may include both keywords in a single function declaration.

PostgreSQL also allows function *overloading* in which the same function name is allowed for different functions, as long as they accept distinct input parameters.



Whenever you drop an existing function, all dependent objects are invalidated and must be recompiled.

Here's an example of a simple SQL function in PostgreSQL:

```
CREATE FUNCTION max_project_nbr
RETURNS int4
AS "SELECT MAX(title_ID) FROM titles AS RESULT"
LANGUAGE 'sql';
```

In this example, we created a UDF that returns the maximum **title_ID** from the **titles** table.

PostgreSQL uses *CREATE FUNCTION* as a substitute for *CREATE PROCEDURE*, as well as to define actions for *CREATE TRIGGER*.

SQL Server

SQL Server supports *CREATE* and *ALTER* for both types of routines. By default, SQL Server stored procedures can return result sets and SQL Server UDFs can return single or multirow result sets using the *TABLE* datatype on *RETURNS* arguments, contrary to the ANSI standard. However, this makes SQL Server routines more flexible and powerful. Use the following syntax to create a user-defined function or stored procedure:

```
CREATE {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]  
( [ {@parameter datatype [VARYING] [= default] [OUTPUT]} [,... ] )  
RETURNS {datatype | TABLE}  
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE |  
    RECOMPILE, ENCRPTION}]  
[FOR REPLICATION]  
[AS]  
    code_block
```

Use the following syntax to alter an existing user-defined function or stored procedure:

```
ALTER {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]  
( [ {@parameter datatype [VARYING] [= default] [OUTPUT]} [,... ] )  
RETURNS {datatype | TABLE}  
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE |  
    RECOMPILE, ENCRPTION}]  
[FOR REPLICATION]  
[AS]  
    code_block
```

Following are the parameter descriptions:

```
CREATE {FUNCTION | PROCEDURE} [owner_name.] object_name[;int]
```

Creates a new UDF or stored procedure in the current database. For SQL Server stored procedures, you may optionally specify a version number in the format *procedure_name;1*, where 1 is an integer indicating the version number. This facility allows you to have multiple versions of a single stored procedure.

```
{@parameter datatype [VARYING] [= default] [OUTPUT]} [,...]
```

Defines one or more input arguments for a UDF or stored procedure. SQL Server parameters are always declared with an at sign (@) as the first character.

where:

VARYING

Used in stored procedures with a *CURSOR* datatype parameter. Indicates that the result set is constructed dynamically by the procedure.

= *default*

Assigns a default value to the parameter. The default value is used whenever the stored procedure or UDF is invoked without supplying a value for the parameter.

OUTPUT

Used for stored procedures, *OUTPUT* is functionally equivalent to the ANSI *OUT* clause in the *CREATE FUNCTION* statement. The value stored in the return parameter is passed back to the calling procedure through the return variables of the SQL Server *EXEC[UTE]* command. Output parameters can be any datatype except *TEXT* and *IMAGE*.

RETURNS { *datatype* | TABLE }

Allows SQL Server UDFs to return a single *datatype* value or to return multiple values via the *TABLE* datatype. The *TABLE* datatype is considered *inline* if it has no accompanying column list and is defined with a single *SELECT* statement. If the *RETURN* clause returns multiple values via the *TABLE* datatype, and if the *TABLE* has defined columns and datatypes, the function is considered a *multistatement, table-valued* function.

WITH

Allows the assignment of additional characteristics to a SQL Server UDF or stored procedure.

ENCRYPTION

Tells SQL Server to encrypt the column in the system table that stores the text of the function, thus preventing unwarranted review of the function code. Usable by both UDFs and stored procedures.

SCHEMABINDING

Specifies that the function is bound to a specific database object, such as a table or view. That database object cannot be altered or dropped as long as the function exists (or maintains the *SCHEMABINDING* option). Usable only by UDFs.

RECOMPILE

Tells SQL Server not to store a cache plan for the stored procedure, but instead to recompile the cache plan each time the stored procedure is executed. This is useful when using atypical or temporary values in the procedure, but can cause serious degradation in performance. Used only by stored procedures. Note that *RECOMPILE* and *ENCRYPTION* can be invoked together.

FOR REPLICATION

Disables execution of the stored procedure on a subscribing server. It is used primarily to create a filtering stored procedure that is executed only by SQL Server's built-in replication engine. It is incompatible with *RECOMPILE*.

Like tables (see *CREATE TABLE*), local and global temporary stored procedures may be declared by prefixing a pound symbol (#) and double-pound symbol (##) to the name of the procedure, respectively. Temporary procedures exist only for the duration of the user or process session that created them. When that session ends, the temporary procedure automatically deletes itself.

A SQL Server stored procedure or UDF may have as many as 1,024 input parameters, specified by the at sign (@). Parameters are defined using SQL Server datatypes. (Parameters of *CURSOR* datatype must be both *VARYING* and *OUTPUT*.) The user or calling process must supply values for any input parameters. However, a default value can be supplied for any input parameter to allow the procedure to execute without a user- or process-supplied value. The default must be a constant or NULL, but it may contain wildcard characters, as discussed later in this chapter in the section [LIKE Operator](#).

SQL Server requires that one or more user-supplied parameters be declared for a given user-defined function. All SQL Server datatypes are supported as parameters, except *TIMESTAMP*. Values returned by the function can be any datatype except *TIMESTAMP*, *TEXT*, *NTEXT*, or *IMAGE*. If an inline table value is required, the *TABLE* option without an accompanying column list may be used.



The *ALTER FUNCTION* and *ALTER PROCEDURE* statements support the full syntax provided by the corresponding *CREATE* statements. You can use the *ALTER* statements to change any of the attributes of an existing routine without changing permissions or affecting any dependent objects.

For UDFs, the *code_block* is either a single *SELECT* statement for an inline function, in the format *RETURN (SELECT...)*, or a series of Transact-SQL statements following the *AS* clause for a multistatement operation. When using *RETURN (SELECT)*, the *AS* clause is optional. Some other rules for SQL Server UDFs:

- When using the *AS* clause, the *code_body* should be encapsulated in *BEGIN...END* delimiters.
- UDFs cannot make any permanent changes to data or cause other lasting side effects. A number of other restrictions exist as a result. For example, *INSERT*, *UPDATE*, and *DELETE* statements may modify only *TABLE* variables local to a function.
- When returning a scalar value, a SQL Server UDF must contain the clause *RETURN datatype*, where *datatype* is the same as that identified in the *RETURNS* clause.
- The last statement of the *code_block* must be an unconditional *RETURN* that returns a single datatype value or *TABLE* value.
- The *code_block* may not contain any global variables that return a perpetually changing value, such as *@@CONNECTIONS* or *GETDATE*. But it may contain those that return a single, unchanging value, such as *@@SERVERNAME*.

The following is an example of a *scalar* function that returns a single value. Once created, the scalar UDF can then be utilized in a query just like a system-supplied function. For example:

```
CREATE FUNCTION metric_volume -- Input dimensions in centimeters.
```

```
    (@length decimal(4,1),
```

```
    @width decimal(4,1),
```

```
    @height decimal(4,1) )
```

```
RETURNS decimal(12,3) -- Cubic Centimeters.
```

```
AS BEGIN
```

```
    RETURN ( @length * @width * @height )
```

```
END
```

```
GO
```

```
SELECT project_name,
```

```
    metric_volume(construction_height,
```

```
    construction_length,
```

```
    construction_width)
```

```
FROM housing_construction
```

```
WHERE metric_volume(construction_height,
```

```
    construction_length,
```

```
    construction_width) >= 300000
```

```
GO
```

An inline table-valued UDF supplies values via a single *SELECT* statement using an *AS RETURN* clause. For example, we can supply a store ID and find all of their **titles**:

```
CREATE FUNCTION stores_titles (@stor_id varchar(30))  
  
RETURNS TABLE  
  
AS  
  
RETURN (SELECT title, qty  
  
        FROM sales AS s  
  
        JOIN titles AS t ON t.title_id = s.title_id  
  
        WHERE s.stor_id = @storeid )
```

Now, let's alter the UDF just a bit by changing the input argument datatype length and adding another condition to the *WHERE* clause (indicated in boldface):

```
ALTER FUNCTION stores_titles  
  
(@stor_id VARCHAR(4))  
  
RETURNS TABLE  
  
AS  
  
RETURN (SELECT title, qty  
  
        FROM sales AS s  
  
        JOIN titles AS t ON t.title_id = s.title_id  
  
        WHERE s.stor_id = @storeid  
  
AND s.city = 'New York')
```

User-defined functions that return *TABLE* values are often selected as result set values or are used in the *FROM* clause of a *SELECT* statement, just as a regular table is used. These *multi-statement, table-valued functions* can have very elaborate code bodies since the *code_block* is composed of many Transact-SQL statements that populate a *TABLE* return variable.

Here is an example invoking a multistatement, table-valued function in a *FROM* clause. Notice that a table alias is assigned, just as for a regular table:

```
SELECT co.order_id, co.order_price  
  
FROM   construction_orders AS co,  
  
       fn_construction_projects('Cancelled') AS fcp  
  
WHERE  co.construction_id = fcp.construction_id  
  
ORDER BY co.order_id  
  
GO
```

For stored procedures, the *code_block* clause contains one or more Transact-SQL commands, up to a maximum size of 128 MB, delimited by *BEGIN* and *END* clauses. Some rules about Microsoft SQL Server stored procedures include:

- The *code_block* allows most valid Transact-SQL statements, but *SET SHOWPLAN_TEXT* and *SET SHOWPLAN_ALL* are prohibited.
- Some other commands have restricted usages within stored procedures, including *ALTER TABLE*, *CREATE INDEX*, *CREATE TABLE*, all *DBCC* statements, *DROP TABLE*, *DROP INDEX*, *TRUNCATE TABLE*, and *UPDATE STATISTICS*.

- SQL Server allows deferred name resolution, meaning that a stored procedure compiles without an error even though it references an object that has not yet been created. SQL Server creates the execution plan and fails only when the object is actually invoked, for instance in a stored procedure, if the object still doesn't exist.
- Stored procedures can be nested easily in SQL Server. Whenever a stored procedure invokes another stored procedure, the system variable @@NESTLEVEL is incremented by 1. It is decreased by 1 when the called procedure completes. Use *SELECT @@NESTLEVEL* inside a procedure or from an ad hoc query session to find the current nesting depth.

In the following example, a SQL Server stored procedure generates a unique 22-digit value (based on elements of the system date and time) and returns it to the calling process:

-- A Microsoft SQL Server stored procedure

```
CREATE PROCEDURE get_next_nbr
    @next_nbr CHAR(22) OUTPUT
AS
BEGIN
    DECLARE @random_nbr INT
    SELECT @random_nbr = RAND( ) * 1000000
    SELECT @next_nbr =
        RIGHT('000000' + CAST(ROUND(RAND(@random_nbr)*1000000,0))
            AS CHAR(6), 6) +
        RIGHT('0000' + CAST(DATEPART (yy, GETDATE( ) )
            AS CHAR(4)), 2) +
        RIGHT('000' + CAST(DATEPART (dy, GETDATE( ) )
            AS CHAR(3)), 3) +
        RIGHT('00' + CAST(DATEPART (hh, GETDATE( ) )
            AS CHAR(2)), 2) +
        RIGHT('00' + CAST(DATEPART (mi, GETDATE( ) )
            AS CHAR(2)), 2) +
        RIGHT('00' + CAST(DATEPART (ss, GETDATE( ) )
            AS CHAR(2)), 2) +
        RIGHT('000' + CAST(DATEPART (ms, GETDATE( ) )
            AS CHAR(3)), 3)
    END
GO
```

See Also

CALL
RETURN

CREATE/ALTER INDEX Statement

Indexes are special objects built on top of tables that speed many data-manipulation operations, such as *SELECT*, *UPDATE*, and *DELETE* statements. The selectivity of a given *WHERE* clause and the available query plans the database query optimizer can choose from is usually based upon the quality of indexes that have been placed on the table in a given database.

The *CREATE INDEX* command is not a part of the ANSI SQL standard, and thus varies greatly among vendors.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

Common Vendor Syntax

```
CREATE [UNIQUE] INDEX index_name ON table_name  
(column_name [, ...])
```

Keywords

```
CREATE [UNIQUE] INDEX index_name
```

Creates a new index named *index_name* in the current database and schema context. Since indexes are associated with specific tables (or sometimes views), the *index_name* need only be unique to the table it is dependent on. The *UNIQUE* keyword defines the index as a unique constraint for the table and disallows any duplicate values into the indexed column or columns of the table. (Refer to [Section 2.4.](#))

```
ON table_name
```

Declares the pre-existing table that the index is associated with. The index is dependent upon the table. If the table is dropped, so is the index.

```
(column_name [, ...])
```

Defines one or more columns in the table that are indexed. The pointers derived from the indexed column or columns enable the database query optimizer to greatly speed up data manipulation operations like *SELECT* and *DELETE* statements. All major vendors support *composite indexes*, also known as *concatenated indexes*. These indexes are used when two or more columns are best searched as a unit—for example, last name and first name.

Rules at a Glance

Indexes are created on tables to speed data manipulation operations that go against the tables such as those in a *WHERE* or *JOIN* clause. Indexes may also speed other operations, like:

- Identifying a *MIN()* or *MAX()* value in an indexed column.
- Sorting or grouping columns of a table.
- Searching based on *IS NULL* or *IS NOT NULL*.
- Fetching data much more quickly when the indexed data is all that is requested. A *SELECT* statement that retrieves data from an index and not directly from the table itself is called a *covering query*. An index that answers a query in this way is a *covering index*.

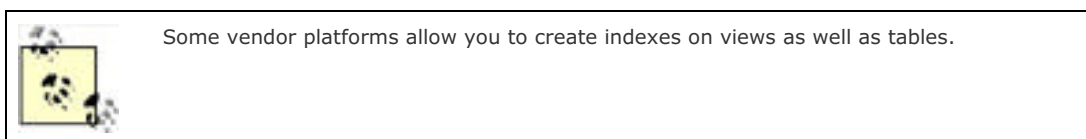
After creating a table, you can create indexes on columns within the table. It is a good idea to create indexes on columns that are frequently part of the *WHERE* clause or *JOIN* clause of the queries that go against a table. For example, the following statement creates an index on a column in the **sales** table that is frequently used in the *WHERE* clause of queries against that table:

```
CREATE INDEX ndx_ord_date ON sales(ord_date);
```

In another case, we want to set up the **pub_name** and **country** as a unique index on the **publishers** table:

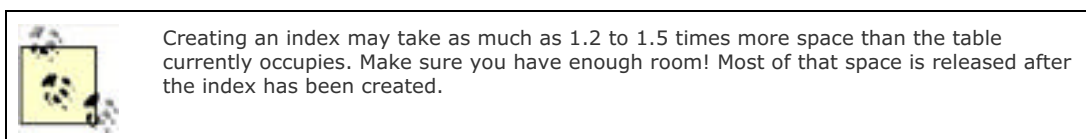
```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country);
```

Since the index is unique, any new record entered into the **publishers** table must have a unique combination of publisher name and country.



Programming Tips and Gotchas

Concatenated indexes are most useful when queries address the columns of the index starting from the left. If you omit left-side columns in a query against a concatenated key, the query may not perform as well. For example, assume that we have a concatenated index on (**last_name**, **first_name**). If we query only by **first_name**, the concatenated index that starts with **last_name** and includes **first_name** may not be any good to us. Some of the vendor platforms have advanced their query engines to the point where this is much less of a problem.



You should be aware that there are situations in which too many indexes can actually slow down system performance. In general, indexes greatly speed lookup operations, especially in *SELECT* statements, against a table or view. However, every index you create adds overhead whenever you perform an *UPDATE*, *DELETE*, or *INSERT* operation because the database must then update all dependent indexes with the values that have changed in the table. As a rule of thumb, six to twelve indexes are about the most you might want to create on a single table.

In addition, indexes take up extra space within the database. The more columns in an index, the more space it consumes. This is not usually a problem, but sometimes catches novices off guard when developing a new database.

Most databases use indexes to create statistical samplings, usually just called *statistics*, so that the query engine can quickly determine which, if any, index or combination of indexes will be most useful for a query. These indexes are always fresh and useful when the index is first created, but may become stale and less useful over time as records are deleted, updated, and inserted within the table. Consequently, indexes, like day-old bread, are not guaranteed to be useful as they age. You need to be sure to refresh, rebuild, and maintain your databases regularly to keep index statistics fresh.

DB2

DB2 supports the *CREATE INDEX* statement, but not the *ALTER INDEX* statement. (DB2 does have the *COMMENT ON INDEX* statement to allow users to update the comment for an index without dropping and recreating the index. Also, DB2 has a *RENAME INDEX* statement to rename an index with having to drop and recreate it.) The *CREATE INDEX* statement has additional clauses that allow partitioning based upon indexed columns, as well as other clauses that impact how data is physically arranged on disk:

```
CREATE [UNIQUE] INDEX index_name  
ON table_name ({column_name [ASC | DESC]} [, ...])
```

[SPECIFICATION ONLY]

[INCLUDE (*column_name* [ASC | DESC] [...])]]

[{CLUSTER | EXTEND USING *index_extension* [(*constant* [...])] }]

[PCTFREE *int*] [MINPCTUSED *int*]

[[DIS]ALLOW REVERSE SCANS]

[COLLECT [SAMPLED [DETAILED]] STATISTICS]

where:

ASC | DESC

Specifies that the values in the index are kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default.

SPECIFICATION ONLY

Used to indicate that the value defined by *table_name* is a nickname; that is, it is the name of a table in a federated database. A DB2 federated database integrates data from heterogeneous data sources, such as tables on Oracle or SQL Server, and names them locally within the DB2 database as a nickname. To tell the DB2 optimizer about indexes on one of these nicknamed Oracle or SQL Server tables, create an index against the nickname and declare *SPECIFICATION ONLY*. DB2 then knows that the index exists and is available to the optimizer, but that it does not have to create the index.

INCLUDE (*column_name* [...])

Adds extra columns to a unique index for added selectivity, but any such included columns are not counted towards uniqueness. The main use for *INCLUDE* columns is to build *covering indexes*. Any columns added with the *INCLUDE* clause count towards the column and byte maximum allowed by DB2. *INCLUDE* is mutually exclusive of *EXTEND USING* and *SPECIFICATION ONLY* clauses, and cannot be used when indexing temporary tables.

CLUSTER

Declares that the index is clustered. A clustered index specifically controls the physical arrangement of the rows in the table to speed query I/O. The cluster factor is maintained and improved dynamically as data is inserted into the table. Only one clustered index is allowed per table. *CLUSTER* is mutually exclusive of *EXTEND USING* and *SPECIFICATION ONLY* clauses, and cannot be used with temporary tables or tables in append mode.

EXTEND USING *index_extension* [(*constant* [...])]

Names an index extension that manages the index. The index must be defined on a single column that is either a structured or a distinct type. One or more *constants* must be provided for any parameters required by the type. *EXTEND* cannot be used on a temporary table. Index extensions are very complex and somewhat uncommon. Refer to the vendor documentation for more detail.

PCTFREE *int*

Specifies a percentage (*int*) of space, which must be an integer of 0 to 99, to leave free on each index page of the index when it is created. When omitted, the default is 10. This option cannot be used on nicknames or temporary tables.

MINPCTUSED *int*

Defines the percentage (*int*) of free space, which must be an integer of 0 to 99, required on an index page before DB2 will consider merging index pages together. DB2 recommends a value of 50 or below so that the database will not spend excessive I/O merging index pages. This clause is usually effective only if background processes are able to gain an occasional table lock. If this is not possible due to heavy usage, consider using the DB2 command *REORG INDEXES*. This option cannot be used on nicknames or temporary tables.

[DIS]ALLOW REVERSE SCANS

Declares that the index can or cannot support scans under the order that the index was created in, with *ALLOW* and *DISALLOW*, respectively. The default is *DISALLOW*. This option cannot be used on nicknames.

COLLECT [SAMPLED [DETAILED]] STATISTICS

Collects index statistics during index creation. If the *SAMPLED* and *DETAILED* clauses are left off, then basic index statistics are created. The *DETAILED* clause collects extra statistics information for the index, namely *PAGE_FETCH_PAIRS* and *CLUSTERFACTOR*. The *SAMPLED* clause tells DB2 that sampling, a shortcut, can be used on extra statistics information from the *DETAILED* clause.

DB2 will not create a new index where an identical index exists. DB2 considers an index identical to another when one index contains the same columns in the same order as the other.

DB2 allows online use of a table while a new index is being created. Read access is always available during the index creation process, but write access is temporarily locked at the end of the index build process. Use the DB2 command *LOCK TABLE* to temporarily lock the table and avoid read access. You should also use the DB2 command *RUNSTATS* to update index statistics whenever a great deal of data has changed or after an index is created without the *COLLECT STATISTICS* clause included in the statement. The following two index creation examples are functionally the same:

-- collect basic index statistics at index creation

```
CREATE INDEX ndx_ord_date ON sales(ord_date) COLLECT STATISTICS;
```

-- collect index statistics after index creation

```
LOCK TABLE sales IN EXCLUSIVE MODE;
```

```
CREATE INDEX ndx_ord_date ON sales(ord_date);
```

```
RUNSTATS ON TABLE sales AND INDEX ndx_ord_date;
```

Now, we'll collect detailed statistics using sampling on the **publishers** table:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
```

```
COLLECT SAMPLED DETAILED STATISTICS;
```

You may wish to create an index on a nicknamed object. In the example below, the **publishers** nickname represents the **current_publishers** table:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
```

```
SPECIFICATION ONLY;
```

The following example shows how to create an extended indextype on a structured type column. The **coordinate_positions** UDT requires a literal value to maintain the **satellite_images** column:

```
CREATE INDEX satellite_images ON gis_sites(coordinates)
```

```
EXTEND USING (coordinate_positions (x'00010010010000100020'));
```



DB2 also supports a statement called *CREATE INDEX EXTENSION*. It is used to enable user-defined indexing on typed tables. Refer to the vendor documentation for more details.

MySQL

MySQL supports a form of the *CREATE INDEX* statement, but not the *ALTER INDEX* statement. MySQL indexes are always stored in B-tree structures on the filesystem. Strings within an index are automatically prefix- and end-space-compressed. MySQL's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE | FULLTEXT] INDEX index_name  
ON table_name (column_name(length) [...])
```

where:

FULLTEXT

Creates a full-text search index against a column. Full-text indexes are only supported on MyISAM table types, *CHAR*, *VARCHAR*, or *TEXT* datatypes, and do not support the (*length*) clause.

MySQL supports the basic industry standard syntax for the *CREATE INDEX* statement. Interestingly, MySQL also lets you build an index on the first *length* characters of a *CHAR* or *VARCHAR* column. MySQL requires the (*length*) clause for *BLOB* and *TEXT* columns. Specifying a length can be useful when selectivity is sufficient in the first, say, 10 characters of a column, and in those situations where saving disk space is very important. For example:

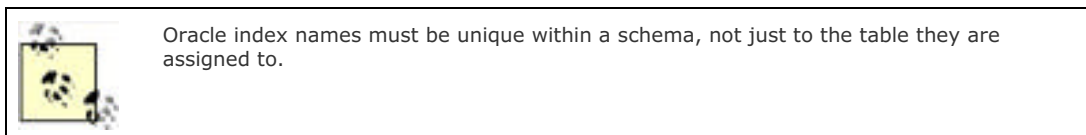
```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name(25),  
country(10))
```

This example indexes only the first 25 characters of the **pub_name** column and the first 10 characters of the **country** column.

As a general rule, MySQL allows at least 16 keys per table with a total maximum length of at least 256 bytes. This can vary by storage engine, however.

Oracle

Oracle allows the creation of indexes on tables, partitioned tables, clusters, and index-organized tables, as well as on scalar type object attributes of a typed table or cluster, and on nested table columns using the *CREATE INDEX* statement. Oracle also allows several types of indexes, including normal B-tree indexes, *BITMAP* indexes (useful for columns that have each value repeated 100 or more times), partitioned indexes, function-based indexes (based on an expression rather than a column value), and domain indexes.



Oracle also supports the *ALTER INDEX* statement. It is used to change or rebuild an existing index without forcing the user to drop and recreate the index. Oracle's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE | BITMAP] INDEX index_name  
{ON  
  {table_name ( {column | expression} [ASC | DESC] [...]) [{INDEXTYPE IS  
  index_type [PARALLEL [inf] | NOPARALLEL] [PARAMETERS ('values') ] |  
  CLUSTER cluster_name |  
  FROM table_name WHERE condition [LOCAL partitioning] }  
[ {LOCAL partitioning | GLOBAL partitioning} ]  
[physical_attributes_clause] [{LOGGING | NOLOGGING}] [ONLINE]  
[COMPUTE STATISTICS] [{TABLESPACE tablespace_name | DEFAULT}]  
[{COMPRESS int | NOCOMPRESS}] [{NOSORT | SORT}] [REVERSE]  
[{PARALLEL [inf] | NOPARALLEL}]
```

The syntax for *ALTER INDEX* is:

```
ALTER INDEX index_name
{ {ENABLE | DISABLE} | UNUSABLE | RENAME TO new_index_name | COALESCE |
[NO]MONITORING USAGE | UPDATE BLOCK REFERENCES |
PARAMETERS ( 'ODCI_params' ) | alter_index_partitioning_clause |
rebuild_clause |
[DEALLOCATE UNUSED [KEEP int [K | M | G | T] ]
[ALLOCATE EXTENT ( [SIZE int [K | M | G | T] ] [DATAFILE 'filename'
[INSTANCE intf ] ]
[SHRINK SPACE [COMPACT] [CASCADE] ]
[{PARALLEL [intf] | NOPARALLEL}]
[{LOGGING | NOLOGGING}]
[physical_attributes_clause] }
```

where the non-ANSI clauses are:

BITMAP

Creates an index bitmap for each index value, rather than indexing each individual row. Bitmaps are best for low concurrency tables, like read-intensive tables. *BITMAP* indexes are incompatible with global partitioned indexes, the *INDEXTYPE* clause, or index-organized tables without a mapping table association.

ASC | DESC

Specifies that the values in the index are kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default. However, be aware that Oracle treats *DESC* indexes as function-based indexes, so there is some difference in functionality between *ASC* indexes and *DESC* indexes. You may not use *ASC* or *DESC* when also using the *INDEXTYPE* clause. *DESC* is ignored on *BITMAP* indexes.

INDEXTYPE IS index_type [PARAMETERS ('*values*')]]

Creates an index on a user-defined type of *index_type*. Domain indexes require that the user-defined type already exists (refer to the [CREATE/ALTER TYPE Statement](#) section). If the user-defined type requires arguments, pass them in using the optional *PARAMETERS* clause. You may also optionally parallelize the creation of the type index using the *PARALLEL* clause (explained below in more detail).

CLUSTER cluster_name

Declares a clustering index based on the specified pre-existing *cluster_name*. On Oracle, a clustering index physically co-locates two tables that are frequently queried on the same columns, usually a primary key and a foreign key. (Clusters are created with the Oracle-specific command *CREATE CLUSTER*.) You do not declare a table or columns on a *CLUSTER* index, since both the tables involved and the columns indexed are already declared with the previously issued *CREATE CLUSTER* statement.

GLOBAL partitioning

Includes the full syntax of:

GLOBAL PARTITION BY

```
{RANGE (column_list) ( PARTITION [partition_name] VALUE LESS THAN
(value_list) [physical_attributes_clause] [TABLESPACE
```

```
tablespace_name] [LOGGING | NOLOGGING] } [,...] ), |  
HASH (column_list) ( PARTITION [partition_name]  
{[TABLESPACE tablespace_name] [[OVERFLOW] TABLESPACE  
tablespace_name] [VARRAY varray_name STORE AS {LOB  
lob_segment_name] [LOB (lob_name) STORE AS [lob_segment_name]  
[TABLESPACE tablespace_name] } } |  
[STORE IN (tablespace_name [,...])] [OVERFLOW STORE IN  
(tablespace_name [,...]) , } [,...] }
```

The *GLOBAL PARTITION* clause declares that the global index is manually partitioned via either range or hash partitioning onto *partition_name*. (The default is to partition the index equally in the same way the underlying table is partitioned, if at all.) You can specify a maximum of 32 columns, though none may be ROWID. You may also apply the *[NO]LOGGING* clause, *TABLESPACE* clause, and the *physical_attributes_clause* (defined earlier) to a specific partition. You cannot partition on ROWID. You may include one or more partitions, along with any attributes, in a comma-delimited list, according to the following:

RANGE

Creates a range-partitioned global index based on the range of values from the table columns listed in the *column_list*.

VALUE LESS THAN (*value_list*)

Sets an upper bound for the current partition of the global index. The values in the *value_list* correspond to the columns in the *column_list*, both of which are comma-delimited lists of columns. Both lists are prefix-dependent, meaning that for a table with columns (*a, b, c*), you could define partitioning on (*a, b*) or (*a, b, c*), but not (*b, c*). The last value in the list should always be the keyword *MAXVALUE*.

HASH

Creates hash-partitioned global index assigned rows in the index to each partition based on a hash function of the values of the columns in the *column_list*. You may specify the exact tablespace to store special database objects such as *VARRAYs*, and *LOBs*, and for any *OVERFLOW* of the specified (or default) tablespaces.

LOCAL *partitioning*

Supports local index partitioning on *range*-partitioned indexes, *list*-partitioned indexes, *hash*-partitioned indexes, and *composite*-partitioned indexes. You may include zero or more partitions, along with any attributes, in a comma-delimited list. When omitted, Oracle generates one or more partitions consistent with those of the table partition. Index partitioning is done in one of three ways:

Range- and list-partitioned indexes

Applied to regular or equipartitioned tables. Range- and list-partitioned indexes (synonyms for the same thing) follow the syntax:

```
LOCAL [ (PARTITION [partition_name]  
{ [physical_attributes_clause] [TABLESPACE tablespace_name]  
[LOGGING | NOLOGGING] |  
[COMPRESS | NOCOMPRESS] } [,...] ) ]
```

All of the options are the same as for *GLOBAL PARTITION* (earlier), except that the scope is for a local index.

Hash-partitioned indexes

Applied to hash-partitioned tables. Hash-partitioned indexes allow you to choose between the earlier syntax and the optional syntax:

```
LOCAL {STORE IN (tablespace_name [...]) } |  
  
(PARTITION [partition_name [TABLESPACE tablespace_name] ] }
```

to store the index partition on a specific tablespace. When you supply more tablespace names than index partitions, Oracle will cycle through the tablespaces when it partitions the data.

Composite-partitioned indexes

Composite-partitioned indexes can be applied on composite-partitioned tables using the following syntax:

```
LOCAL [STORE IN (tablespace_name [...]) ]  
  
PARTITION [partition_name  
  
{physical_attributes_clause [TABLESPACE tablespace_name]  
  
[LOGGING | NOLOGGING] |  
  
[COMPRESS | NOCOMPRESS] }  
  
[ {STORE IN (tablespace_name [...]) } |  
  
(SUBPARTITION [subpartition_name [TABLESPACE  
  
tablespace_name]) } ]
```

You may use the *LOCAL STORE* clause shown under the hash-partitioned indexes entry or from the *LOCAL* clause under the range- and list-partitioned indexes entry. (When using the *LOCAL* clause, substitute the keyword *SUBPARTITION* for *PARTITION*.)

physical_attributes_clause

Establishes values for one or more of the following settings: *PCTFREE int* (discussed here), *INITTRANS int*, and *STORAGE* (discussed in the [CREATE/ALTER TABLE Statement](#) section). When omitted, Oracle defaults to *PCTFREE* 10 and *INITTRANS* 2. The default for *MAXTRANS* depends on the system block size, according to the following:

PCTFREE int

Designates the percentage of free space to leave on each block of the index as it is created. This speeds up new entries and updates on the table. However, *PCTFREE* is only applied when the index is created. It is not maintained over time. Therefore, the amount of free space can erode over time as records are inserted, updated, and deleted from the index. This clause is not allowed on index-organized tables.

INITTRANS int

Designates the initial number of concurrent transactions allocated to each data block of the database. The value may range from 1 to 255. If a *physical_attributes_clause* is omitted, Oracle defaults to *PCTFREE* 10 and *INITTRANS* 2.

LOGGING | NOLOGGING

Tells Oracle to log the creation of the index on the redo logfile (*LOGGING*), or to not log it (*NOLOGGING*). This clause also sets the default behavior for subsequent bulk loads using Oracle SQL* Loader. For partitioned indexes, this clause establishes the default value of all partitions, segments associated with the partitions, and the defaults used on any partitions or subpartitions later added with an *ALTER TABLE...ADD PARTITION* statement. (When using *NOLOGGING*, we recommend that you take a full backup after the index has been loaded in case the index must be rebuilt due to a failure.)

ONLINE

Allows data manipulation on the table while the index is being created. Even when *ONLINE*, there is a very small window at the end of the create index operation where the table will be locked while the index creation completes. Any changes made to the base table at that time will then be reflected in the newly created index. *ONLINE* is incompatible with bitmap, cluster, or parallel clauses. It also cannot be used on indexes on a **UROWID** column or on index-organized tables with more than 32 columns in their primary keys.

COMPUTE [STATISTICS]

Collects statistics while the index is created, when it can be done at relatively little cost. Otherwise, you will have to collect statistics after the index is created.

*TABLESPACE *tablespace_name* | DEFAULT*

Assigns the index to a specific tablespace. When omitted, the index is placed on the default tablespace. Use the *DEFAULT* keyword to explicitly place an index into the default tablespace. (When local partitioned indexes are placed in *TABLESPACE DEFAULT*, the index partition (or subpartition) is placed in the corresponding tablespace of the base table partition (or subpartition).

*COMPRESS [*int*] | NOCOMPRESS*

Enables or disables key compression, respectively. Compression eliminates repeated occurrences of key column values, yielding a substantial space savings at the cost of speed. The integer value, *int*, defines the number of prefix keys to compress. The value can range from 1 to the number of columns in the index for nonunique indexes, and from 1 to *n*-1 columns for unique indexes. The default is *NOCOMPRESS*, but if you specify *COMPRESS* without an *int* value, the default is *COMPRESS n* (for nonunique indexes) or *COMPRESS n-1* (for unique indexes), where *n* is the number of columns in the index. *COMPRESS* can only be used on nonpartitioned and nonbitmapped indexes.

NOSORT | REVERSE

NOSORT an index to be created quickly for a column that is already sorted in ascending order. If the values of the column are not in perfect ascending order, the operation aborts, allowing a retry without the *NOSORT* option. *REVERSE*, by contrast, places the index blocks in storage by reverse order (excluding *rowid*). *REVERSE* is mutually exclusive of *NOSORT* and cannot be used on a bitmap index or an index-organized table. *NOSORT* is most useful for creating indexes immediately after a base table is loaded with data in pre-sorted order.

*PARALLEL [*int*] | NOPARALLEL*

Allows the parallel creation of the index using multiple server process, each operating on a distinct subset of the index, to speed up the operation. An optional integer value, *int*, may be supplied to define the exact number of parallel threads used in the operation. When omitted, Oracle calculates the number of parallel threads to use. *NOPARALLEL*, the default, causes the index to be created serially.

{ENABLE | DISABLE}

Enables or disables a pre-existing function-based index, respectively. You cannot specify any other clauses of the *ALTER INDEX* statement with *ENABLE* or *DISABLE*.

UNUSABLE

Marks the index (or index partition or subpartition) as unusable. When *UNUSABLE*, an index (or index partition or subpartition) may only be rebuilt or dropped and recreated before it can be used.

*RENAME TO *new_index_name**

Renames the index from *index_name* to *new_index_name*.

COALESCE

Merges the contents of index blocks used to maintain the index-organized table so that the blocks can be reused. *COALESCE* is similar to *SHRINK*, though *COALESCE* compacts the segments less densely than does *SHRINK*, and it does not release unused space.

[NO]MONITORING USAGE

Declares that Oracle should clear existing information on index usage and monitor the index, posting information in the *V\$OBJECT_USAGE* dynamic performance view, until *ALTER INDEX ... NOMONITOR USAGE* is executed.

UPDATE BLOCK REFERENCES

Updates all stale guess data block addresses stored as part of the index row on normal or domain indexes of an index-organized table. The guess data blocks contain the correct database addresses for the corresponding blocks identified by the primary key. This clause cannot be used with other clauses of the *ALTER INDEX* statement.

PARAMETERS ('*ODCI_params*')

Specifies a parameter string passed, without interpretation, to the ODCI indextype routine of a domain index. The parameter string, called '*ODCI_params*', may be up to 1,000 characters long. Refer to the vendor documentation for more information on ODCI parameter strings.

alter_index_partitioning_clause

Refer to [Oracle partitioned and subpartitioned tables](#) in the [CREATE/ALTER TABLE Statement](#) section for more details.

rebuild_clause

Rebuilds the index or a specific partition (or subpartition) of the index. A successful rebuild marks an *UNUSABLE* index as *USABLE*. The syntax for the *rebuild_clause* is:

REBUILD {[NO]REVERSE | [SUB]PARTITION *partn_name*}

[{PARALLEL [*int*] | NOPARALLEL}] [TABLESPACE *tablespace_name*]

[PARAMETERS ('*ODCI_params*')] [ONLINE] [COMPUTE STATISTICS]

[COMPRESS *int* | NOCOMPRESS] [[NO]LOGGING]

[*physical_attributes_clause*]

where:

[NO]REVERSE

Stores the bytes of the index block in reverse order and excludes rows when the index is rebuilt (*REVERSE*) or stores the bytes of the index blocks in regular order (*NOREVERSE*).

DEALLOCATE UNUSED [KEEP *int* [K | M | G | T]]

Deallocates unused space at the end of the index (or at the end of each range- or hash-partition of a partitioned index) and frees the space for other segments in the tablespace. The optional *KEEP* keyword defines how many bytes, *int*, above the high-water mark the index will keep after deallocation. You can append a suffix to the *int* value to indicate that the value equals kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). When the *KEEP* clause is omitted, all unused space is freed.

ALLOCATE EXTENT ([SIZE *int* [K | M | G | T]] [DATAFILE '*filename*'] [INSTANCE *int*])

Explicitly allocates a new extent for the index using the specified parameters. You may mix and match any of the parameters. *SIZE* specifies the amount of next extent. The size of the extents may be in bytes (no suffix), kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). *DATAFILE* allocates an entirely new datafile to the index extent. *INSTANCE*, used only on Oracle Real Application Clusters, makes a new extent available to a freelist group associated with the specified instance.

SHRINK SPACE [COMPACT] [CASCADE]

Shrinks the index segments, though only segments in tablespaces with automatic segment management may be shrunk. Shrinking a segment moves rows in the table, so make sure *ENABLE ROW MOVEMENT* is also used in the *ALTER TABLE...SHRINK* statement. Oracle compacts the segment, releases the emptied space, and adjusts the high-water mark unless the optional keywords *COMPACT* and/or *CASCADE* are applied. The *COMPACT* keyword only defragments the segment space and compacts the index, but it does not readjust the high-water mark nor empty the space immediately. The *CASCADE* keyword performs the same shrinking operation (with some restrictions and exceptions) on all dependent objects of the index. The statement *ALTER INDEX ... SHRINK SPACE COMPACT* is functionally equivalent to *ALTER INDEX ... COALESCE*.

By default, Oracle indexes are non-unique. It is also important to know that Oracle's regular B-tree indexes do not include records that have a null key value.

Oracle does not support indexes on columns with the following datatypes: *LONG*, *LONG RAW*, *REF* (with the *SCOPE* attribute), or user-defined datatypes. You may create indexes on functions and expressions, but they cannot allow null values or aggregate functions. When you create an index on a function, the function, if it has no parameters, should show an empty set. For example: *function_name()*. If the function is a UDF, it must be *DETERMINISTIC*.

Oracle supports a special index structure called an *index-organized table* (IOT). IOTs combine the table data and primary key index on a single physical structure, instead of having separate structures for the table and for the index. IOTs are created using the *CREATE TABLE...ORGANIZATION INDEX* statement. Refer to [CREATE/ALTER TABLE Statement](#) for more information on making an IOT.

Oracle automatically creates any additional indexes on an index-organized table as secondary indexes. Secondary indexes do not support the *REVERSE* clause.

Oracle allows the creation of partitioned indexes and tables with the *PARTITION* clause. Consequently, Oracle's indexes also support partitioned tables. The *LOCAL* clause tells Oracle to create separate indexes for each partition of a table. The *GLOBAL* clause tells Oracle to create a common index for all the partitions.

Note that any time an object name is referenced in the syntax diagram, you may optionally supply the schema. This applies to indexes, tables, etc., but not to tablespaces. You must have the explicitly declared privilege to create an index in a schema besides the current one.

As an example, you can use a statement such as the following to create an Oracle index that is compressed, created in parallel, and with compiled statistics, but without logging the creation:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
COMPRESS 1 PARALLEL NOLOGGING COMPUTE STATISTICS;
```

As with other Oracle object creation statements, you can control how much space they consume and in what increments they grow. The following example constructs an index in Oracle on a specific tablespace with specific instructions for how the data is to be stored:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
STORAGE (INITIAL 10M NEXT 5M PCTINCREASE 0)
TABLESPACE publishers;
```

For example, when you create the **housing_construction** table as a partitioned table on an Oracle server, you should also create a partitioned index with its own index partitions:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
ON housing_construction(project_id)
GLOBAL PARTITION BY RANGE (project_id)
(PARTITION part1 VALUES LESS THAN ('H')
TABLESPACE construction_part1_ndx_ts,
PARTITION part2 VALUES LESS THAN ('P')
TABLESPACE construction_part2_ndx_ts,
PARTITION part3 VALUES LESS THAN (MAXVALUE)
TABLESPACE construction_part3_ndx_ts);
```

If in fact the **housing_construction** table used a composite partition, we could accommodate that here:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
ON housing_construction(project_id)
STORAGE (INITIAL 10M MAXEXTENTS UNLIMITED)
LOCAL (PARTITION part1 TABLESPACE construction_part1_ndx_ts,
PARTITION part2 TABLESPACE construction_part2_ndx_ts
(SUBPARTITION subpart10, SUBPARTITION subpart20,
SUBPARTITION subpart30, SUBPARTITION subpart40,
SUBPARTITION subpart50, SUBPARTITION subpart60),
PARTITION part3 TABLESPACE construction_part3_ndx_ts);
```

In the following example, we could rebuild the **project_id_ind** index that was created earlier by using parallel execution processes to scan the old index and build the new index in reverse order:

```
ALTER INDEX project_id_ind
REBUILD REVERSE PARALLEL;
```

Similarly, we could split out an additional partition on **project_id_ind**:

```
ALTER INDEX project_id_ind
SPLIT PARTITION part3 AT ('S')
INTO (PARTITION part3_a TABLESPACE constr_p3_a LOGGING,
PARTITION part3_b TABLESPACE constr_p3_b);
```

PostgreSQL

PostgreSQL allows the creation of ascending-order indexes, as well as *UNIQUE* indexes. Its implementation also includes a performance enhancement under the *USING* clause. PostgreSQL's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE] INDEX index_name ON table_name
[USING {BTREE | RTREE | HASH} ]
{function_name | (column_name [,...]) }
[WHERE condition]
```

where:

USING [BTREE | RTREE | HASH | GIST]

Specifies one of three dynamic access methods to optimize performance. Of particular importance is the fact that the indexes are fully dynamic and do not require periodic refreshes of their statistics. The *USING* options are as follows:

BTREE

Uses Lehman-Yao's high-concurrency B-tree structures to optimize the index. This is the default method when no other is specified. B-tree indexes, the default, can be invoked for comparisons using =, <, <=, >, and >=. B-tree supports multicolumn indexes.

RTREE

Uses Guttman's quadratic-split algorithm R-tree structures to optimize the index. R-tree indexes can be invoked for comparison using <<, &<, &>, >>, @, ~=, and &&. R-tree indexes must be single-column indexes.

HASH

Uses Litwin's linear hashing algorithm to optimize the index. Hash indexes can be invoked for comparisons using =. Hash indexes must be single-column indexes.

GIST

Uses Generalized Index Search Trees (GISTs) to optimize the index. GIST indexes may be multicolumn.

function_name

Defines a user-defined function, rather than a column from the base table, as the basis of the index values. A function-based index is mutually exclusive of a regular column-based index.

WHERE condition

Defines a *WHERE* clause search condition, which is then used to generate a *partial* index. A partial index contains entries for a select set of records in the table, not all records of the table. You can get some interesting effects from this clause. For example, you can pair *UNIQUE* and *WHERE* to enforce uniqueness for a subset of the table rather than for the whole table. The *WHERE* clause must:

- Reference columns in the base table (though they need not be columns of the index itself).
- Not make use of aggregate functions.
- Not use subqueries.

In PostgreSQL, a column may have an associated *operator class* based on the datatype of the column. An operator class specifies the operators for a particular index. Although users are free to define any valid operator class for a given column, the default operator class is the appropriate operator class for that column type.

In the following example, we create an index using the *GIST* indextype. We also make sure that uniqueness is only enforced for publishers outside of the USA:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
USING GIST
WHERE country <> 'USA';
```

SQL Server

SQL Server's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE] [[NON]CLUSTERED] INDEX index_name
ON {table_name | view_name} (column [ASC | DESC] [,...])
[WITH [PAD_INDEX]
    [,] FILLFACTOR = int]
    [,] IGNORE_DUP_KEY]
    [,] DROP_EXISTING]
    [,] STATISTICS_NORECOMPUTE ]
    [,] SORT_IN_TEMPDB ]
[ON filegroup]
```

where:

[NON]CLUSTERED

Controls the physical ordering of data for the table using either a *CLUSTERED* or *NONCLUSTERED* index. The columns of a *CLUSTERED* index determine the order in which the records of the table are physically written. Thus, if you create an ascending clustered index on column A of table **Foo**, the records will be written to disk in ascending alphabetical order. The *NONCLUSTERED* clause, the default, when either value is omitted, creates a secondary index containing only pointers and has no impact on how the actual rows of the table are written to disk.

ASC | DESC

Specifies that the values in the index are kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default.

WITH

Allows the specification of one or more optional attributes for the index.

PAD_INDEX

Specifies that space should be left open on each 8K index page, according to the value established by the *FILLFACTOR* setting.

FILLFACTOR = int

Declares a percentage value, *int*, from 1 to 100 that tells SQL Server how much of each 8K data page should be filled at the time the index is created. This is useful to reduce I/O contention and page splits when a data page fills up. Creating a clustered index with an explicitly defined fillfactor can increase the size of the index, but also speed up processing in certain circumstances.

IGNORE_DUP_KEY

Controls what happens when a duplicate record is placed into a unique index through an insert or update operation. If this value is set for a column, only the duplicate row is excluded from the operation. If this value is not set, then all records in the operation (even nonduplicate records) are rejected as duplicates.

DROP_EXISTING

Drops any pre-existing indexes on the table and rebuilds the specified index.

STATISTICS_NORECOMPUTE

Stops SQL Server from recomputing index statistics. This can speed the *CREATE INDEX* operation, but it may mean that the index is less effective.

SORT_IN_TEMPDB

Stores any intermediate results used to build the index in the system database, *TEMPDB*. This increases the space needed to create the index, but could speed processing if *TEMPDB* is on a different disk than the table and index.

ON filegroup

Creates the index on a given pre-existing filegroup. This enables the placing of indexes on a specific hard disk or RAID device. Issuing a *CREATE CLUSTERED INDEX...ON FILEGROUP* effectively moves a table to the new filegroup, since the leaf level of the clustered index is the same as the actual data pages of the table.

SQL Server allows the creation of a unique clustered index on a view, effectively materializing the view. This can greatly speed up data retrieval operations against the view. Once a view has a unique clustered index, nonclustered indexes also can be added to the view. Note that the view also must be created using the *SCHEMABINDING* option. Indexed views are only allowed in SQL Server 2000 Enterprise Edition, unless you add a *NOEXPAND* hint to the view. Indexed views support data retrieval, but not data modification. (See the [SQL Server](#) section of [CREATE/ALTER VIEW Statement](#) for specific restrictions on indexed views.)

SQL Server allows up to 249 nonclustered indexes (unique or non-unique) on a table, as well as one primary key index. Index columns may not be of datatype *NTEXT*, *TEXT*, or *IMAGE*. Concatenated keys may contain up to 16 columns and/or a total of 900 bytes across all fixed-length columns.

SQL Server automatically parallelizes the creation of the index according to the configuration option *max degree of parallelism*.

It is often necessary to build indexes that span several columns—i.e., a *concatenated key*. Here is an example:

```
CREATE UNIQUE INDEX project2_ind
ON housing_construction(project_name, project_date)
WITH PAD_INDEX, FILLFACTOR = 80
ON FILEGROUP housing_fg
GO
```

Adding the *PAD_INDEX* clause and setting the *FILLFACTOR* to 80 tells SQL Server to leave the index and data pages 80% full, rather than 100% full. The example also tells SQL Server to create the index on the **housing_fg** filegroup, rather than the default filegroup.

See Also

CREATE/ALTER TABLE
DROP

CREATE/ALTER METHOD Statement

The *CREATE/ALTER METHOD* statements allow the creation of a new database method or the alteration of an already existing database method. An easy (but loose) way to think of a method is that it is a user-defined function associated with a user-defined type. For example, a method called **Office**, of type **Address**, can accept a *VARCHAR* input parameter and passes out a result of **Address**.

An implicitly defined method is created every time a structured type is created (see [CREATE/ALTER TYPE Statement](#) for more details). Using a combination of the *CREATE TYPE* statement and the *CREATE METHOD* statement creates user-defined methods.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
{CREATE | ALTER} [INSTANT | STATIC] METHOD method_name
( [ [{IN | OUT | INOUT}] param datatype [AS LOCATOR] [RESULT] [...]] )
RETURNS datatype
```

FOR *udt_name*

[SPECIFIC *specific_name*]

code_body

Keywords

CREATE | ALTER [INSTANT | STATIC] *method_name*

Creates a new method or alters an existing method and, optionally, specifies it as an *INSTANT* or *STATIC* method.

[(IN | OUT | INOUT] *param datatype* [,...])

Declares one or more parameters to be passed into the method in a comma-delimited list enclosed in parentheses. Parameters used with a method may pass a value *IN*, *OUT*, or both in and out via *INOUT*. The syntax for the parameter declaration is:

[{IN | OUT | INOUT}] *parameter_name1 datatype*,

[{IN | OUT | INOUT}] *parameter_name2 datatype*,[...]

Make sure the name is unique within the method. When used with *ALTER*, this clause adds parameters to a pre-existing method. Refer to [Chapter 2](#) for details on datatypes.

AS *LOCATOR*

The optional *AS LOCATOR* clause is used to validate an external routine with a *RETURNS* parameter that is a *BLOB*, *CLOB*, *NCLOB*, *ARRAY*, or user-defined type. In other words, the locator (i.e., a pointer) for the *LOB* is returned but not the entire value of the *LOB*.

RESULT

Designates a user-defined type, but is not needed for standard datatypes.

RETURNS *datatype*

Declares the datatype of the results returned by a method. The key purpose of a user-defined method is to return a value. If you need to change the datatype of a *RETURNS* parameter on the fly, use the *CAST* clause (refer to the function *CAST*). For example, *RETURNS VARCHAR(12) CAST FROM DATE*.

FOR *udt_name*

Associates the method with a specific, pre-existing user-defined type, created using *CREATE TYPE*.

SPECIFIC *specific_name*

Uniquely identifies the function generally used with user-defined types.

Rules at a Glance

User-defined methods are essentially a different approach to the same output provided by user-defined functions. For example, consider the following two pieces of code:

```
CREATE FUNCTION my_fcn (order_udt)
RETURNS INT;
CREATE METHOD my_mthd ( )
RETURNS INT
FOR order_udt;
```

Although the code sections for the function and method are different, they do exactly the same thing.

The rules for use and invocation of methods are otherwise the same as for functions.

Programming Tips and Gotchas

The main difficulty with methods is that they are an object-oriented approach to the same sort of functionality provided by user-defined functions. Since they accomplish the same work, using a different approach, it can be hard to decide which approach to use.

DB2

DB2 supports both the *CREATE METHOD* and *ALTER METHOD* statements:

```
CREATE [SPECIFIC] METHOD method_name
( [ [{IN | OUT | INOUT}] [param] datatype [AS LOCATOR] [RESULT] [...]] )
[RETURNS datatype [{AS LOCATOR | CAST FROM datatype [AS LOCATOR]} ] ]
FOR udt_name
EXTERNAL [NAME 'name'] [TRANSFORM GROUP group_name]
code_body
```

The syntax for *ALTER METHOD* is quite a bit simpler:

```
ALTER METHOD method_name
FOR udt_name
EXTERNAL NAME 'name'
```

where:

[SPECIFIC] METHOD *method_name*

Defines the method being created or altered. The optional *SPECIFIC* keyword identifies a method using the name assigned (either implicitly or explicitly) using the *CREATE TYPE* statement. The method must exist in the named or current schema.

CAST FROM *datatype*

Returns a different datatype to the invoking statement from the datatype that was returned by the method.

EXTERNAL [NAME '*external_name*'] [TRANSFORM GROUP *group_name*]

Indicates that the statement is registering an external method, rather than creating an internal (SQL) method. The user-defined type created using the *CREATE TYPE* statement must be defined with a *LANGUAGE* other than SQL.

NAME '*name*'

Defines the name of the external code that implements the method. The name may be up to 254 characters long. The name must be appropriate for the type of *LANGUAGE* associated with the user-defined type. This clause may also define the name of an internal (SQL) method specification for user-defined types defined with *LANGUAGE SQL*.

TRANSFORM GROUP *group_name*

Specifies a transform group for user-defined structure type transformations when the method is invoked. When omitted, the transform group *DB2_FUNCTION* is the default.

The method specification is determined by the *LANGUAGE* setting of the *CREATE TYPE* statement associated with the method. For *LANGUAGE SQL*, the method specification is SQL. For any other *LANGUAGE* setting, the method specification is external.

The following example method, called **office_zip**, checks to see if a ZIP Code is valid based on the user-defined type **address_type**:

```
CREATE METHOD office (address address_type)
RETURNS INTEGER
FOR address_type
RETURN
(CASE
  WHEN (self..zip = address.zip) THEN 1 -- valid
  ELSE 0 -- invalid
END);
```

In the next example, the method **distance** uses an external routine and transform group to process the between two **address_types**:

```
CREATE METHOD distance (address_type)
FOR address_type
EXTERNAL NAME 'addrlib!distance'
TRANSFORM GROUP method_group;
```

Later, we could change the method so that it uses a different external routine:

```
ALTER METHOD distance ( )
FOR TYPE address_type
EXTERNAL NAME 'newaddrlib!distance_2';
```

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

CREATE/ALTER TYPE

CREATE ROLE Statement

CREATE ROLE allows the creation of a named set of privileges that may be assigned to users of a database. When a role is granted to a user, that user gets all the privileges and permissions of that role. Roles are generally accepted as one of the best means for maintaining security and controlling privileges within a database.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
CREATE ROLE role_name [WITH ADMIN {CURRENT_USER | CURRENT_ROLE}]
```

Keywords

```
CREATE ROLE role_name
```

Creates a new role and differentiates that role from a host DBMS user and other roles. A role can be assigned any permission that a user can be assigned. The important difference is that a role can then be assigned to one or more users, thus giving them all the permissions of that role.

```
WITH ADMIN {CURRENT_USER | CURRENT_ROLE}
```

Assigns a role immediately to the currently active user or currently active role along with the privilege to pass the use of the role on to other users. By default, the statement defaults to *WITH ADMIN CURRENT_USER*.

Rules at a Glance

Using roles for database security can greatly ease administration and user maintenance. The general steps for using roles in database security are:

1. Assess the needs for roles and pick the role names (e.g., *administrator*, *manager*, *data_entry*, *report_writer*, etc.).
2. Assign permissions to each role as if it were a standard database user using the *GRANT* statement. For example, the *manager* role might have permissions to read and write from all user tables in a database, while the *report_writer* role only has permissions to execute read-only reports against certain reporting tables in the database.
3. Use the *GRANT* statement to grant roles to users of the system according to the types of work they will perform.

Permissions can be disabled using the *REVOKE* command.

Programming Tips and Gotchas

The main problem with roles is that occasionally a database administrator will provide redundant permissions to a role and separately to a user. If you ever need to prevent a user's access to the resource in situations like this, you usually will need to *REVOKE* the permissions twice. The role must be revoked from the user and then the specific user-level privilege must also be revoked from the user.

DB2

Not supported.

Instead, DB2 uses operating system groups, so you can achieve similar functionality but you need to:

1. Create an OS group.
2. Grant or revoke permissions to the group.
3. Add users to the group at the OS level.

DB2 allows OS groups to be assigned special admin roles at the instance level using the **sysadm_group**, **sysctrl_group**, and **sysmaint_group** configuration parameters.

MySQL

Not supported.

Oracle

Although it is not currently supported by the ANSI standard, Oracle also offers an *ALTER ROLE* statement. Oracle supports the concept of roles, though Oracle's implementation of the commands is very different from the ANSI SQL standard:

```
{CREATE | ALTER} ROLE role_name  
  
[NOT IDENTIFIED |  
  
IDENTIFIED {BY password | EXTERNALLY | GLOBALLY |  
  
USING package_name}]
```

where:

```
CREATE | ALTER ROLE role_name
```

Identifies the name of the new role being created or the pre-existing role being modified.

```
NOT IDENTIFIED
```

Declares that no password is needed for the role to receive authorization to the database. This is the default.

```
IDENTIFIED
```

Declares that the users of the role must authenticate themselves by the method indicated before the role is enabled via the *SET ROLE* command, where:

```
BY password
```

Creates a local role authenticated by the string value of *password*. Only single-byte characters are allowed in the password even when using a multibyte character set.

```
EXTERNALLY
```

Creates an external role that is authenticated by the operating system or a third-party provider. In either case, the external authentication service will likely require a password.

GLOBALLY

Creates a global role that is authenticated by an enterprise directory service, such as an LDAP directory.

USING *package_name*

Creates an application role that enables a role only through an application that uses a PL/SQL package of *package_name*. If you omit the schema, Oracle assumes that the package is in your schema.

In Oracle, the role is created first, then granted privileges and permissions as if it is a user via the *GRANT* command. When users want to get access to the permissions of a role protected by a password, they use the *SET ROLE* command. If a password is placed on the role, any user wishing to access it must provide the password with the *SET ROLE* command.

Oracle ships with several preconfigured roles. *CONNECT*, *DBA*, and *RESOURCE* are available in all versions of Oracle. *EXP_FULL_DATABASE* and *IMP_FULL_DATABASE* are newer roles used for import and export operations. The *GRANT* statement reference has a more detailed discussion of all of the preconfigured roles available in Oracle.

The following example uses *CREATE* to specify a new role in Oracle, *GRANTS* it privileges, assigns it a password with *ALTER ROLE*, and *GRANTS* that role to a couple of users:

```
CREATE ROLE boss;
```

```
GRANT ALL ON employee TO boss;
```

```
GRANT CREATE SESSION, CREATE DATABASE LINK TO boss;
```

```
ALTER ROLE boss IDENTIFIED BY le_grande_fromage;
```

```
GRANT boss TO emily, jake;
```

PostgreSQL

Although PostgreSQL does not support the *CREATE ROLE* command, it offers a nearly identical facility of its own called *CREATE GROUP*. PostgreSQL also offers an *ALTER GROUP* statement to modify existing groups. In PostgreSQL, a group and a role are identical:

```
{CREATE | ALTER} GROUP name
```

```
[ [WITH] {SYSID int | USER username [,...]} ]
```

```
[ {DROP | ADD} username [,...]} ]
```

where:

```
{CREATE | ALTER} GROUP name
```

Specifies if a new group (i.e., role) is created or if an existing group is modified where *name* is the name of the group to create or modify.

```
WITH {SYSID int | USER username [,...]} ]
```

Assigns an explicit group ID to the new group, with the *SYSID* clause, or assigns a list of one or more pre-existing users to the group. When omitted, PostgreSQL will create the group using the highest assigned group ID plus one (starting at 1). The *WITH* clause is only used with the *CREATE GROUP* statement.

```
{DROP | ADD} username
```

Drops or adds one or more pre-existing users to the group. Used only with the *ALTER GROUP* clause.

Use the *DROP GROUP* clause to drop a group you no longer want.

SQL Server

Microsoft SQL Server does not support the *CREATE ROLE* command, but has the equivalent capability via the system stored procedure **sp_add_role**.

See Also

GRANT
REVOKE

CREATE SCHEMA Statement

This statement creates a *schema*—i.e., a named group of related objects. A schema is a collection of tables, views, and permissions granted to specific users or roles. According to the ANSI standard, specific object permissions are not schema objects in themselves and do not belong to a specific schema. However, roles are sets of privileges that do belong to a schema.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

CREATE SCHEMA [*schema_name*] [AUTHORIZATION *owner_name*]

[DEFAULT CHARACTER SET *char_set_name*]

[PATH *schema_name* [,...]]

[ANSI CREATE *statements* [...]]

[ANSI GRANT *statements* [...]]

Keywords

CREATE SCHEMA [*schema_name*]

Creates a schema called *schema_name*. When omitted, the database will create a schema name for you using the name of the user who owns the schema.

AUTHORIZATION *owner_name*

Sets the *owner_name* as the user who owns the schema. When omitted, the current user is set as the owner. The ANSI standard allows you to omit either the *schema_name* or the *AUTHORIZATION* clause, or to use them both together.

DEFAULT CHARACTER SET *char_set_name*

Declares a default character set of *char_set_name* for all objects created within the schema.

PATH *schema_name* [,...]

Optionally declares a file path and filename for any unqualified routines (i.e., stored procedures, user-defined functions, user-defined methods) in the schema.

ANSI CREATE *statements* [...]

Contains one or more *CREATE* statements. No commas are used between the *CREATE* statements.

ANSI GRANT *statements* [...]

Contains one or more *GRANT* statements to previously defined objects. Usually, the objects were created earlier in the same *CREATE SCHEMA* statement, but they may be any previously created object. No commas are used between the *GRANT* statements.

Rules at a Glance

The *CREATE SCHEMA* statement is a container that can hold many other *CREATE* and *GRANT* statements. The most common way to think of a schema is as all of the objects that a specific user owns. For example, the user **jake** may own several tables and views including the table, **publishers**, in his own schema. Meanwhile, the user **dylan** may own several other tables and views in his schema, but may also own his own separate copy of the **publishers** table.

The ANSI standard requires that all *CREATE* statements are allowed in a *CREATE SCHEMA* statement. In practice, most implementations of *CREATE SCHEMA* only allow three subordinate statements: *CREATE TABLE*, *CREATE VIEW*, and *GRANT*. The order of the commands is not important, meaning you can grant privileges on tables or views whose *CREATE* statements appear later in the *CREATE SCHEMA* statement.

Programming Tips and Gotchas

It is not required, but is considered a best practice, to arrange the objects and grants within a *CREATE SCHEMA* statement in an order that will execute naturally. In other words, *CREATE VIEW* statements should follow the *CREATE TABLE* statements that they depend on, and then *GRANT* statements should come last.

If your database system uses schemas, we recommend that you always reference your objects by schema and then object name (as in **jake.publishers**). If you do not include a schema qualifier, the database platform will typically assume the default schema for the current user connection.

Some database platforms do not explicitly support the *CREATE SCHEMA* command. However, they implicitly create a schema when a user creates database objects. For example, Oracle creates a schema whenever a user is created. The *CREATE SCHEMA* command is simply a single-step method of creating all the tables, views, and other database objects along with their permissions.

DB2

DB2 supports the basic elements of the *CREATE SCHEMA* statement. It *does not* support the *PATH* or *DEFAULT CHARACTER SET* clauses:

```
CREATE SCHEMA [schema_name] [AUTHORIZATION owner_name]
```

```
[ ANSI CREATE statements [...] ]
```

```
[ ANSI GRANT statements [...] ]
```

The syntax and usage is otherwise identical to the ANSI standard.

DB2 creates a schema implicitly whenever a new schema is referenced in a *GRANT* or *CREATE* statement.

DB2 does not support an *ALTER SCHEMA* statement.

MySQL

Not supported.

Oracle

In Oracle, the *CREATE SCHEMA* does not create a schema—only the *CREATE USER* statement does that. *CREATE SCHEMA* allows a user to perform multiple *CREATE*s and *GRANT*s in a previously created schema in one SQL statement:

```
CREATE SCHEMA AUTHORIZATION schema_name
```

[ANSI CREATE *statements* [...]]

[ANSI GRANT *statements* [...]]

Note that Oracle only allows ANSI standard *CREATE TABLE*, *CREATE VIEW*, and *GRANT* statements within a *CREATE SCHEMA* statement. You *should not* use any of Oracle's extensions to these commands when using the *CREATE SCHEMA* statement.

The following Oracle example places the permissions before the objects within the *CREATE SCHEMA* statement:

```
CREATE SCHEMA AUTHORIZATION emily

GRANT SELECT, INSERT ON view_1 TO sarah

GRANT ALL ON table_1 TO sarah

CREATE VIEW view_1 AS

    SELECT column_1, column_2

    FROM table_1

    ORDER BY column_2

CREATE TABLE table_1(column_1 INT, column_2 CHAR(20));
```

As the example above shows, the order of the statements within the *CREATE SCHEMA* statement is unimportant. Oracle commits the *CREATE SCHEMA* statement only if all *CREATE* and *GRANT* statements in the statement complete successfully.

PostgreSQL

Not supported.

SQL Server

SQL Server supports the basic *CREATE SCHEMA* statement without support for the *PATH* clause or the *DEFAULT CHARACTER SET* clause:

```
CREATE SCHEMA AUTHORIZATION owner_name
```

[ANSI CREATE *statements* [...]]

[ANSI GRANT *statements* [...]]

If any statement fails within the *CREATE SCHEMA* statement, then the entire statement fails.

SQL Server does not require that the *CREATE* or *GRANT* statements be in any particular order, except that nested views must be declared in their logical order. Thus, if **view_100** references **view_10**, then **view_10** must appear in the *CREATE SCHEMA* statement before **view_100**.

For example:

```
CREATE SCHEMA AUTHORIZATION katie

GRANT SELECT ON view_10 TO public

CREATE VIEW view_10(col1) AS SELECT col1 FROM foo

CREATE TABLE foo(col1 INT)

CREATE TABLE foo

    (col1 INT PRIMARY KEY,

    col2 INT REFERENCES foo2(col1))
```

```
CREATE TABLE foo2
(
  col1 INT PRIMARY KEY,
  col2 INT REFERENCES foo(col1))
```

See Also

CREATE/ALTER TABLE
CREATE/ALTER VIEW
GRANT

CREATE/ALTER TABLE Statement

Manipulating tables is one of the most common activities that database administrators and programmers perform when working with database objects. This section details how to create and modify tables.

The ANSI standard represents a sort of least common denominator among the vendors, though not every option of the ANSI standard version of *CREATE TABLE* and *ALTER TABLE* is offered by each of the vendors. However, the ANSI standard does represent the basic form that can be used across all of the platforms. In turn, the vendor platforms offer a variety of extensions and additions to the ANSI standards for *CREATE* and *ALTER TABLE*.



Typically, a great deal of consideration goes into the design and creation of a table. This discipline is known as *database design*. The discipline of analyzing the relationship of a table to its own data and to other tables within the database is known as *normalization*. We recommend that database programmers and administrators alike study both database design and normalization principles thoroughly before issuing *CREATE TABLE* commands.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 statement *CREATE TABLE* creates a permanent or temporary table within the database where the command is issued. The syntax is as follows:

```
CREATE [{LOCAL TEMPORARY} | GLOBAL TEMPORARY} TABLE table_name
```

```
(column_name datatype attributes [,...] ) |
```

```
[column_name WITH OPTIONS options] |
```

```
[LIKE table_name] |
```

```
[REF IS column_name
```

```
{SYSTEM GENERATED | USER GENERATED | DERIVED} ]
```

```
[CONSTRAINT constraint_type [constraint_name] [,...] ]
```

```
[OF type_name [UNDER super_table] [table_definition] ]
```

[ON COMMIT {PRESERVE ROWS | DELETE ROWS}]

The SQL2003 statement *ALTER TABLE* allows many useful modifications to be made to an existing table without dropping any existing indexes, triggers, or permissions assigned to it. Following is the *ALTER TABLE* syntax:

```
ALTER TABLE table_name
[ADD [COLUMN] column_name datatype attributes]
| [ALTER [COLUMN] column_name SET DEFAULT default_value]
| [ALTER [COLUMN] column_name DROP DEFAULT]
| [ALTER [COLUMN] column_name ADD SCOPE table_name]
| [ALTER [COLUMN] column_name DROP SCOPE {RESTRICT | CASCADE}]
| [DROP [COLUMN] column_name {RESTRICT | CASCADE}]
| [ADD table_constraint]
| [DROP CONSTRAINT table_constraint_name {RESTRICT | CASCADE}]
```

Keywords

[LOCAL | GLOBAL] TEMPORARY

Declares a permanent table or a *TEMPORARY* table of *LOCAL* or *GLOBAL* scope. *Local temporary tables* are only accessible to the session that created them and are automatically dropped when the session that created them terminates. *Global temporary tables* are accessible by all active sessions, but are automatically dropped when the session that created them terminates. Do not qualify a temporary table with a schema name.

(*column_name datatype attributes* [,...])

Defines a comma-delimited list of one or more columns, their datatypes, and any additional attributes such as nullability. Every table declaration must contain at least one column, which may include:

Column_name

Specifies a name for a column. The name must be a valid identifier according to the rules of the specific DBMS. Make sure the name makes sense!

Datatype

Associates a specific datatype with the column of *column_name*. An optional *length* may be specified for datatypes that allow user-defined lengths, for example *VARCHAR(255)*. It must be a valid datatype on the specific DBMS. Refer to [Chapter 2](#) for a full discussion of acceptable datatypes and vendor variations.

Attributes

Associates specific constraint attributes with the *column_name*. Many attributes may be applied to a single *column_name*, no commas required. Typical ANSI attributes include:

NOT NULL

Tells the column to reject null values or accept null values, when *NOT NULL* is omitted. Any *INSERT* or *UPDATE* statement that attempts to place a null value in a *NOT NULL* column will fail and roll back.

DEFAULT expression

Tells the column to use the value of *expression* when no value is supplied by an *INSERT* or *UPDATE* statement. The expression must be acceptable according to the datatype of the column; for example, no alphabetic characters may be used in an *INTEGER* column. *Expression* may be a string or numeric literal, but you may also define a user-defined function or system function. SQL2003 allows these system functions in a *DEFAULT* expression: *NULL*, *USER*, *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *LOCALTIME*, *CURRENT_TIMESTAMP*, *LOCALTIMESTAMP*, *ARRAY*, or *ARRAY[]*.

COLLATE collation_name

Defines a specific collation, or sort order, for the column it is assigned to. The name of the collation is platform dependent. If you do not define a collation, the columns of the table default to the collation of the character set used for the column.

REFERENCES ARE [NOT] CHECKED [ON DELETE {RESTRICT | SET NULL}]

Indicates whether references are to be checked on a *REF* column defined with a scope clause. The optional *ON DELETE* clause tells whether any values in records referenced by a deleted record should set to *NULL* or whether the operation should be restricted.

CONSTRAINT constraint_name [constraint_type [constraint]]

Assigns a constraint and, optionally a constraint name, to the specific column. Constraint types are discussed in [Chapter 2](#). Because the constraint is associated with a specific column, the constraint declaration assumes that the column is the only one in the constraint. After the table is created, the constraint is treated as a table-level constraint.

column_name [WITH OPTIONS options]

Defines a column with special options, such as a scope clause, a default clause, a column-level constraint, or a *COLLATE* clause. For many implementations, the *WITH OPTIONS* clause is restricted to the creation of typed tables.

LIKE table_name

Creates a new table with the same column definitions as the pre-existing table of *table_name*.

REF IS column_name {SYSTEM GENERATED | USER GENERATED | DERIVED}

Defines the object identifier column (OID) for a typed table. An OID is necessary for the root table of a table hierarchy. Based on the option, the *REF* might be automatically generated by the system (*SYSTEM GENERATED*), manually provided by the user when inserting the row (*USER GENERATED*), or derived from another *REF* (*DERIVED*). Requires the inclusion of a *REFERENCES* column attribute for *column_name*.

CONSTRAINT constraint_type [constraint_name] [,...]

Assigns one or more constraints to the table. This option is noticeably different than the *CONSTRAINT* option at the column level because column-level constraints are assumed to apply only to the column they are associated with. Table-level constraints, however, give the option of associating multiple columns with a constraint. For example, in a sales table you might wish to declare a unique constraint on a concatenated key of **store_id**, **order_id**, and **order_date**. This can only be done using a table-level constraint. Refer to [Chapter 2](#) for a full discussion of constraints.

OF type_name [UNDER super_table] [table_definition]

Declares that the table is based upon a pre-existing user-defined type (see [CREATE/ALTER TYPE Statement](#)). In this situation, the table may have only a single column for each attribute of the structured type, plus an additional column as defined in the *REF IS* clause. The *REF* datatype is described in greater detail in the section [CREATE/ALTER TYPE Statement](#). This clause is incompatible with the *LIKE table_name* clause.

where:

UNDER super_table [table_definition]

Declares the direct supertable of the currently declared table within the same schema, if any exists. The table being created is thus a direct subtable of the supertable. You may optionally provide a complete *table_definition* of the new subtable, complete with columns, constraints, etc.

ON COMMIT {PRESERVE ROWS | DELETE ROWS}

ON COMMIT PRESERVE ROWS preserves data rows in a temporary table on issuance of a *COMMIT* statement.
ON COMMIT DELETE ROWS deletes all data rows in a temporary table on *COMMIT*.

ADD [COLUMN] *column_name datatype attributes*

Adds a column to a table, along with the appropriate datatype and attributes.

ALTER [COLUMN] *column_name* SET DEFAULT *default_value*

Adds a default value to the column if none exists, and resets the default value if a previous one exists.

ALTER [COLUMN] *column_name* DROP DEFAULT

Completely removes a default from the named column.

ALTER [COLUMN] *column_name* ADD SCOPE *table_name*

Adds a *scope* to the named column. A scope is a reference to a user-defined datatype (see [CREATE/ALTER TYPE Statement](#)”).

ALTER [COLUMN] *column_name* DROP SCOPE [RESTRICT | CASCADE]

Drops a scope from the named column. *RESTRICT* and *CASCADE* clauses are defined at the end of this list.

DROP COLUMN *column_name* [RESTRICT | CASCADE]

Drops the named column from the table. *RESTRICT* and *CASCADE* clauses are defined at the end of this list.

ADD *table_constraint*

Adds a table constraint of the specified name and characteristics to the table.

DROP CONSTRAINT *constraint_name* [RESTRICT | CASCADE]

Drops a previously defined constraint from the table.

RESTRICT

Tells the DBMS to abort the command if it finds any other objects in the database that depend on the object.

CASCADE

Tells the DBMS to drop any other objects that depend on the object.

Rules at a Glance

The typical *CREATE TABLE* statement is very simple. Generally, it names the table and any columns contained in the table. Many table definitions also include a nullability constraint for most of the columns, as in this SQL Server example:

CREATE TABLE housing_construction

(project_number INT NOT NULL,

project_date DATE NOT NULL,


```
project_name    VARCHAR(50) NOT NULL,  
construction_color  NCHAR(20) ,  
construction_height DECIMAL(4,1),  
construction_length DECIMAL(4,1),  
construction_width  DECIMAL(4,1),  
construction_volume INT      )
```

This example adds a foreign key to the example table:

```
-- Creating a column-level constraint
```

```
CREATE TABLE favorite_books  
(isbn          CHAR(100) PRIMARY KEY,  
book_name     VARCHAR(40) UNIQUE,  
category      VARCHAR(40) ,  
subcategory   VARCHAR(40) ,  
pub_date      DATETIME  NOT NULL,  
purchase_date DATETIME  NOT NULL,  
             CONSTRAINT fk_categories FOREIGN KEY (category)  
             REFERENCES category(cat_name));
```

The foreign key on the **categories** column relates it to the **cat_name** table in the **category** table. All the vendors mentioned in this book support this syntax.



Examples for creating a table with each constraint type are shown in [Chapter 2](#). Similarly, the foreign key could be added after the fact as a multicolumn key including both the **category** and **subcategory** columns:

```
ALTER TABLE favorite_books ADD CONSTRAINT fk_categories  
             FOREIGN KEY (category, subcategory)  
             REFERENCES category(cat_name, subcat_name));
```

Now, we could use an *ALTER TABLE* statement to drop the constraint altogether:

```
ALTER TABLE favorite_books DROP CONSTRAINT fk_categories RESTRICT;
```

Listed here are more full examples from **pubs**, the sample database that ships with Microsoft SQL Server and Sybase Adaptive Server:

```
-- For a Microsoft SQL Server database
```

```
CREATE TABLE jobs  
(job_id SMALLINT IDENTITY(1,1) PRIMARY KEY CLUSTERED,  
job_desc VARCHAR(50) NOT NULL DEFAULT 'New Position',  
min_lvl TINYINT NOT NULL CHECK (min_lvl >= 10),
```

```
max_lvl TINYINT NOT NULL CHECK (max_lvl <= 250))
-- For a MySQL database
CREATE TABLE employee
(emp_id INT AUTO_INCREMENT CONSTRAINT PK_emp_id PRIMARY KEY,
fname VARCHAR(20) NOT NULL,
minit CHAR(1) NULL,
lname VARCHAR(30) NOT NULL,
job_id SMALLINT NOT NULL DEFAULT 1
REFERENCES jobs(job_id),
job_lvl TINYINT DEFAULT 10,
pub_id CHAR(4) NOT NULL DEFAULT ('9952')
REFERENCES publishers(pub_id),
hire_date DATETIME NOT NULL DEFAULT (CURRENT_DATE( ));
CREATE TABLE publishers
(pub_id char(4) NOT NULL
CONSTRAINT UPKCL_pubind PRIMARY KEY CLUSTERED
CHECK (pub_id IN ('1389', '0736', '0877', '1622', '1756')
OR pub_id LIKE '99[0-9][0-9]'),
pub_name varchar(40) NULL,
city varchar(20) NULL,
state char(2) NULL,
country varchar(30) NULL DEFAULT('USA'))
```

Once you get into the vendor extensions, the *CREATE TABLE* command is no longer portable between database platforms. The following is an example of an Oracle *CREATE TABLE* statement with many storage properties:

```
CREATE TABLE classical_music_cds
(music_id INT,
composition VARCHAR2(50),
composer VARCHAR2(50),
performer VARCHAR2(50),
performance_date DATE DEFAULT SYSDATE,
duration INT,
cd_name VARCHAR2(100),
CONSTRAINT pk_class_cds PRIMARY KEY (music_id)
USING INDEX TABLESPACE index_ts
STORAGE (INITIAL 100K NEXT 20K),
CONSTRAINT uq_class_cds UNIQUE
(composition, performer, performance_date)
```

```
    USING INDEX TABLESPACE index_ts  
  
    STORAGE (INITIAL 100K NEXT 20K)  
  
TABLESPACE tabledata_ts;
```

When issuing a *CREATE* or *ALTER* statement, we recommend that it be the only statement in your transaction. For example, do not attempt to create a table and select from it in the same batch. Instead, (1) create the table, (2) verify the operation, (3) *COMMIT*, and then (4) perform any subsequent operations against the table.

Table_name is the name for a new or existing table. New table names should start with an alphabetic character and in general contain no other special symbol besides underscore (`_`). Length of the name and exact composition is governed differently according to each vendor.

When creating or altering a table, the list of column definitions is always encapsulated within parentheses and separated by commas between each new column definition.

Programming Tips and Gotchas

The user issuing the *CREATE TABLE* command must have the appropriate permissions. Similarly, any user wishing to *ALTER* or *DROP* a table should own the table within his own schema or have adequate permissions to the alter or drop the table. Since the ANSI standard does not specify the privileges required, expect some variation between vendors.

You can encapsulate a *CREATE TABLE* or *ALTER TABLE* statement within a transaction, using a *COMMIT* or *ROLLBACK* statement to explicitly conclude the transaction. We recommend that it be the *only* command in the transaction.

You can have a great deal of control over the way that the records of a table are physically written to disk through extensions to the ANSI standard. DB2 and SQL Server use a technique called *clustered indexes* to control the way that records are written to disk. Oracle uses a technique that is functionally similar called an *index-organized table*.

Some databases will lock a table that is being modified by an *ALTER TABLE* statement. It is wise to issue this command only on tables that are not in use on a production database.

Some databases will lock the target *and* source tables when using the *CREATE TABLE...LIKE* statement.

DB2

DB2 supports a very rich and elaborate set of extensions to the *CREATE TABLE* statement:

```
CREATE TABLE table_name  
  
    {(column_name datatype attributes [,...]) } |  
  
    OF type_name [{UNDER super_table INHERIT SELECT PRIVILEGES |  
    HIERARCHY hierarchy_name}] [table_definition] |  
  
    AS subquery {derived_table_options |  
    DEFINITION ONLY [copy_options] } |  
  
    [LIKE table_name] [copy_options] |  
  
    [ [(staging_column [,...]) ]  
    FOR table_name PROPAGATE IMMEDIATE] }  
  
[ORGANIZE BY [DIMENSIONS] (column_name [,...]) ]  
  
[DATA CAPTURE {NONE | CHANGES} ]  
  
[IN tablespace_name [INDEX IN tablespace_name]  
    [LONG IN tablespace_name]]  
  
[VALUE COMPRESSION]  
  
[WITH RESTRICT ON DROP]  
  
[NOT LOGGED INITIALLY]  
  
[{{REPLICATED | PARTITIONING KEY (column_name [,...])
```

```
[USING HASHING] } ]  
[OPTIONS (REMOTE_SERVER 'server_name'  
[, REMOTE_SCHEMA 'schema_name'  
[, REMOTE_TABNAME 'table_name' ] ]
```

Similarly rich, the DB2 statement *ALTER TABLE* allows many useful modifications to be made to an existing table without dropping any existing indexes, triggers, or permissions assigned to it:

```
ALTER TABLE table_name  
{[ADD { {[COLUMN] column_name datatype attributes } |  
  table_constraint | partitioning_key | RESTRICT_ON_DROP } |  
{[ALTER { {CHECK | FOREIGN KEY} constraint_name constraint_definition } |  
  [COLUMN] column_name column_definition } |  
{[DROP constraint_type constraint_name] | partitioning_key |  
  RESTRICT_ON_DROP } |  
[DATA CAPTURE {NONE | CHANGES [INCLUDE LONGVAR COLUMNS]} |  
[ACTIVATE NOT LOGGED INITIALLY [WITH EMPTY TABLE] ] |  
[PCTFREE int] |  
[LOCKSIZE {ROW | TABLE} ] |  
[APPEND {ON | OFF} ] |  
[ [NOT] VOLATILE [CARDINATIVITY] ] |  
[ [DE]ACTIVATE VALUE COMPRESSION] } [...] |  
SET MATERIALIZED QUERY AS {DEFINITION ONLY | subquery}
```

The parameters for both commands are as follows:

(*column_name datatype attributes* [,...])

Defines columns according to the ANSI standard. However, DB2 supports several additional attributes that are specific to special datatypes on DB2. All the ANSI standard attributes, such as *NOT NULL*, are supported. Refer to the section on datatypes in [Chapter 2](#) for more details.

UNDER *super_table* INHERIT SELECT PRIVILEGES | HIERARCHY *hierarchy_name*

Similar to the ANSI standard *OF* clause in the *CREATE TABLE* command, with two added optional clauses — *INHERIT* and *HIERARCHY*. *INHERIT* tells DB2 that any user or group with read privileges on the *super_table* should also have read privileges on the new subtable. *HIERARCHY* associates the subtable with a table hierarchy, where *hierarchy_name* is the name of the root table of the hierarchy. When omitted, DB2 will generate its own *hierarchy_name*.

table_definition

Defines the elements of the typed table, including addition options for columns, an *OID* column on the root table, and table constraints.

AS *subquery*

Materializes the table from the results of a query. You may include a standard column definition list, but the columns must match the number and datatype of columns retrieved by the subquery.

derived_table_options

The full syntax for *derived_table_options* is:

DATA INITIALLY DEFERRED REFRESH {DEFERRED | IMMEDIATE}

[{ENABLE | DISABLE} QUERY OPTIMIZATION]

[MAINTAINED BY {SYSTEM | USER}]

The keywords have the following meanings:

DATA INITIALLY DEFERRED

Defers the insertion of data into the table until a DB2 *REFRESH TABLE* command is issued.

REFRESH {DEFERRED | IMMEDIATE}

Allows the table to be refreshed only when a DB2 *REFRESH TABLE* statement is issued (via *DEFERRED*) or immediate after any data manipulation operation (via *IMMEDIATE*).

{ENABLE | DISABLE} QUERY OPTIMIZATION

Enables or disables the derived table for query optimization when DB2 decides optimization is appropriate. Of course, the table is always available for direct queries.

MAINTAINED BY {SYSTEM | USER}

Declares that the derived table data is maintained automatically by the *SYSTEM* (the default) or manually by the *USER* via the DB2 *REFRESH TABLE* command. If you select *MAINTAINED BY USER*, the table must also be declared as *REFRESH DEFERRED*.

DEFINITION ONLY [*copy options*]

Defines the table without populating the table with a query result set.

copy_options

Allows the modification of the *DEFINITION ONLY* and *LIKE* clauses. Copy options include:

{INCLUDING | EXCLUDING} [COLUMN] DEFAULTS

Includes or excludes column defaults of the copied table, respectively. *INCLUDING* is the default for base tables or temporary tables. Otherwise, *EXCLUDING* is the default.

{INCLUDING | EXCLUDING} [IDENTITY] [COLUMN ATTRIBUTES]

Includes or excludes column attributes of the copied table, respectively. The *IDENTITY* attributes of the column may optionally be copied.

LIKE *table_name* [*copy options*]

Copies the entire set of column definitions from an existing table, *table_name*, though a view or nickname is also allowed. You may use the **copy_options** shown previously in the list to alter the default behavior of the *LIKE* clause.

[(*staging_column* [,...])] FOR *table_name* PROPAGATE IMMEDIATE]

Defines an intermediate table (i.e., a staging table) between the base table(s) that define a derived table and

the derived table itself. (Naturally, this clause should only be used with the *AS subquery* clause.)

The components of this clause are:

staging_column

Defines one or more staging columns. However, the list of staging columns will always contain two more columns than the number of columns in the derived table — *GLOBALTRANSID* and *GLOBALTRANSTIME*. The *PROPAGATE* clause, adds one more column — *OPERATIONTYPE* — which is used to maintain consistency in replicated a materialized query table. When omitted, DB2 will create the appropriate staging columns it needs for the operation.

FOR *table_name*

Names the table that defines the staging table. *table_name* must represent a derived table with *REFRESH DEFERRED* already existing in the database. Whenever a *REFRESH TABLE* command is issued, the contents of the staging table are used to refresh the derived table.

PROPAGATE IMMEDIATE

Specifies that data modifications to the underlying tables cascade immediately to the staging table.

ORGANIZE BY [DIMENSIONS] (*column_name* [,...])

Defines a cluster or dimension on a column or group of columns, depending on whether the *DIMENSIONS* keyword is excluded or included, respectively. In effect, a clustered index is defined with the *ORGANIZE BY* clause, while the *ORGANIZE BY DIMENSIONS* clause is used to create a clustered index on more than one column (i.e., on a dimension).

DATA CAPTURE {NONE | CHANGES}

Writes extra data to the log in support of inter-database replication. *NONE* declares that no extra information will be stored, while *CHANGES* declares that changes are written to the log. The *DATA CAPTURE CHANGES* clause is required on replicated tables. This clause is incompatible when creating subtables.

IN *tablespace_name* [INDEX IN *tablespace_name*][LONG IN *tablespace_name*]

Declares a specific, pre-existing tablespace where the table is placed. This clause is incompatible when creating a subtable and in System Managed tablespaces. The components of the clause are:

INDEX IN *tablespace_name*

Declares a separate pre-existing tablespace where the indexes of the table will be placed. When omitted, the default is the same tablespace as the table data.

LONG IN *tablespace_name*

Declares a separate pre-existing tablespace where the data of long datatypes (*LONG VARCHAR*, *LONG VARGRAPHIC*, *LOB*, etc.) for the table will be placed. When omitted, the default is the same tablespace as the table data.

VALUE COMPRESSION

Enables high compression storage of NULL and 0-length data values.

WITH RESTRICT ON DROP

Restricts the table and its tablespace from being dropped.

NOT LOGGED INITIALLY

Declares that any changes to the data of the table that occur in the same batch as when the table is created are not logged. On the other hand, all catalog changes and storage changes are logged. This setting can cause the rollback of an entire batch of work under certain circumstances.

REPLICATED

Declares that the data in the table is replicated on each database partition of the database partition group where it is defined, in effect copying the data to all database partitions. This clause is incompatible with *PARTITIONING KEY*, but is required for derived tables.

PARTITIONING KEY (*column_name* [,...]) [USING HASHING]

Defines a partitioning key for the table where *column_name* becomes the partitioning key. The *column_name* may not be a long or special datatype — *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK* — or structured or distinct types built upon these datatypes. This clause is incompatible when creating subtables. When omitted and the table is defined on a database partition group, DB2 will attempt to find a method for partitioning the table. Otherwise, the table will be placed on a single-partition database partition group. *USING HASHING* declares that a hashing algorithm is used to distribute data among the partitions.

OPTIONS (REMOTE_SERVER '*server_name*' [, REMOTE_SCHEMA '*schema_name*'] [, REMOTE_TABNAME '*table_name*']

Defines a remote operation, such as creating, altering, or dropping a table on a federated system. This is called *transparent DDL*. Federated systems are quite complex and are outside of the ANSI standard. Refer to the vendor documentation for more information.

ADD *table_constraint*

Adds a table-level constraint to an existing table.

ALTER {CHECK | FOREIGN KEY} *constraint_name constraint_definition*

Alters the attributes of an existing *CHECK* or *FOREIGN KEY* of *constraint_name* and assigns it a new *constraint_definition*. You may also apply these additional constraint characteristics:

[NOT] ENFORCED

Changes the constraint to *ENFORCED* or *NOT ENFORCED*. Under *ENFORCED*, the constraint is automatically enforced by the database. *NOT ENFORCED* causes the constraint to not be checked during data modification operations.

{ENABLE | DISABLE} QUERY OPTIMIZATION

Enables or disables query optimization on the constraint.

ALTER [COLUMN] *column_name column_definition*

Alters a specific column, *column_name*, in the table. You can change a number of column characteristics beyond the standard column definition options, including things like *SCOPE*, *COMPRESS*, and identity alterations. Refer to the material later in this section for information on type-related attributes and other alterable attributes. Some additional characteristics can be changed on a column-by-column basis, such as:

SET DATA TYPE VARCHAR (*int*)

Alters the length of the named *VARCHAR* column to *int*. The value of *int* should not increase the total row length or the total primary key length beyond the allowed maximum.

SET EXPRESSION AS (*generation-expression*)

Alters the original generated expression, a sort of auto-incrementing identity column, of the named column to a new *generation-expression*. The difference between a generated expression

and an identity is that identities are automatically handled by DB2, but generated expressions give you full control over the values returned by the expression. Generated expressions can be quite complex and are covered in the vendor documentation. This clause requires the table be placed in "check pending state," where the data in the table is not verified against constraints, via DB2's *SET INTEGRITY* statement. Afterwards, the values must all be checked.

SET INLINE LENGTH *int*

Alters the inline length of the named structured type column to *int*, storing the values inline with the rest of the data in the row.

DROP *constraint_type constraint_name*

Drops a constraint of *constraint_name* from the table. The *constraint_type* may be any of the allowable types: primary key, foreign key, unique, check, as well as *PARTITIONING KEY*. You may also substitute *constraint_type* with *RESTRICT ON DROP*, to drop the *RESTRICT ON DROP* characteristic.

DATA CAPTURE {NONE | CHANGES [INCLUDE LONGVAR COLUMNS] }

Captures extra information to the log in support of replication. Usable with any non-typed table. The options for *DATA CAPTURE* are:

NONE

Specifies that no extra information is captured.

CHANGES

Writes extra information to the table in support of replication. This clause is required for replication. This clause is incompatible with tables that store data on non-catalog partitions. The *INCLUDE LONGVAR COLUMNS* clause includes *LONG VARCHAR* and *LONG VARCHAR* columns in the data replication.

ACTIVATE NOT LOGGED INITIALLY [WITH EMPTY TABLE]

Disables logging of any transaction made in the same batch as the *ALTER TABLE* statements. Statements that are not logged include *INSERT*, *DELETE*, *UPDATE*, *CREATE INDEX*, *DROP INDEX*, or another *ALTER TABLE* statement. Logging returns to normal after the current SQL batch completes. The optional *WITH EMPTY TABLE* clause tells DB2 to empty the table of data. The data is not recoverable except through the DB2 *RECOVER* facility.

PCTFREE *int*

Declares a percentage of free space, *int*, that DB2 should leave on each page during load or reorganization of the table.

LOCKSIZE {ROW | TABLE}

Declares that locks should be taken at a *ROW* or *TABLE* level when reading or writing data in the table. The default is *ROW*. This clause is only supported in the root table of a typed-table hierarchy.

APPEND {ON | OFF}

Adds all new data to the end of the table when turned *ON* or puts data in the free space available within data pages throughout the table when turned *OFF*. The default is *OFF*. Setting this value to *ON* causes DB2 not to keep any free space information. The *APPEND* clause is not usable on tables with a clustered index.

[NOT] VOLATILE [CARDINALITY]

Declares whether the table cardinality varies greatly over time. When set to *VOLATILE*, tells DB2 to generally ignore statistics in favor of an index scan because the amount of data in the table change a great deal. *VOLATILE* is supported only on normal tables and the root table of a typed-table hierarchy.

NOT VOLATILE tells DB2 that the amount of data in the table is generally static, so that it can use existing statistics and optimization paths. Adding the optional *CARDINALITY* keyword tells DB2 that the amount of data in the table may change but that the table itself is static.

[DE]ACTIVATE VALUE COMPRESSION

Deactivates or activates the compression of NULL and zero-length data values in the table for all datatypes of *BLOB*, *CLOB*, *DBLOB*, *LONG VARCHAR*, or *LONG VARGRAPHIC*.

SET MATERIALIZED QUERY {AS DEFINITION ONLY | *subquery*}

Alters the properties of a materialized query table. *DEFINITION ONLY* tells DB2 to consider the table as a normal table and not a materialized query table. When using *DEFINITION ONLY*, the materialized query table cannot be replicated. The *subquery* clause is any valid subquery used to define a materialized query table. The *subquery* redefines a regular table as a materialized query table during query optimization. When using the *subquery* clause, the table may not be a typed table, have constraints, unique indexes, or triggers, or be defined as or referenced in a materialized query table.

DB2 also has a number of special attributes that may be applied to columns over the standard constraints that the ANSI SQL standard provides. These optional attributes are specific to specific datatypes. The following are *LOB*-related attributes:

[NOT] LOGGED

Declares whether changes made to the column are written to the logfile (*LOGGED*) or not (*NOT LOGGED*). The default is *LOGGED*. However, any *LOB* values greater than 10 megabytes should not be logged and those over 1 gigabyte cannot be logged.

[NOT] COMPACT

Compresses the values of *LOBs* as much as possible, usually by freeing any extra space on the last group used by the *LOB* or allowing free space in the last group to go unused. *COMPACT* takes some extra processing and I/O overhead. *NOT COMPACT* is the default.

For example, you might have a table called **emp_images**, which stores the security ID images of your employees:

```
CREATE TABLE emp_images
(emp_id      INT NOT NULL,
 emp_photo_image LONGVARGRAPHIC NOT LOGGED COMPACT)
```

DB2 supports the following datalink-related attributes:

LINKTYPE URL

Defines a URL as the type of link.

NO LINK CONTROL

Declares that there is no check for file existence.

FILE LINK CONTROL [*options*]

Declares that file existence will be checked. Additional options may impact the style of check, including:

INTEGRITY [ALL]

Sets the level of checks between the *DATALINK* value and an actual file. The optional *ALL* keyword disables operating system control for deleting or renaming the file.

READ PERMISSION {FS | DB}

Sets read permission on the file as either filesystem derived (*FS*) or database derived (*DB*).

WRITE PERMISSION {FS | BLOCKED | ADMIN}

Sets write permission on the file as either filesystem derived (*FS*), no writes allowed (*BLOCKED*), or determined by the Data Links Manager (*ADMIN*). The *WRITE PERMISSION ADMIN* clause is based on a passed token. You should further specify *[NOT] REQUIRING TOKEN FOR UPDATE*, which details whether a write token is needed to finish a file update operation.

RECOVERY {YES | NO}

Declares that DB2 will support point in time recovery of files referenced by the values in the column (*YES*) or not (*NO*). *RECOVERY YES* also requires the *INTEGRITY ALL* clause and either *WRITE PERMISSION BLOCKED* or *WRITE PERMISSION ADMIN*.

ON UNLINK {RESTORE | DELETE}

Declares that DB2 either return the file to its original permissions when the *DATALINK* value is changed or deleted (*RESTORE*) or that the file is deleted when unlinked (*DELETE*). *ON UNLINK RESTORE* requires the *INTEGRITY ALL* clause and either *WRITE PERMISSION BLOCKED* or *WRITE PERMISSION ADMIN*. *ON UNLINK DELETE* requires the *READ PERMISSION DB* clause and either *WRITE PERMISSION BLOCKED* or *WRITE PERMISSION ADMIN*.

MODE DB2OPTIONS

Defines a default group of file link options: *INTEGRITY ALL*, *READ PERMISSION FS*, *WRITE PERMISSION FS*, and *RECOVERY NO*.

DB2 supports the following type-related attributes:

SCOPE type_name

Declares the scope of a reference-type column, where *type_name* is the pre-existing typed table or view. It is required for columns that will be used as the left operand of a dereference operator or as the argument of a *DEREF* function.

INLINE LENGTH int

Declares a maximum byte size, *int*, for a structured-type column. If the structured type is too long, DB2 will automatically store the structured types in a separate table from the base table, similar to the way *LOB* values are stored.

DB2 supports just one storage-related attribute:

COMPRESS SYSTEM DEFAULT

Defines that the system default value for any column is compressed. It should not be used on columns of *DATE*, *TIME*, or *TIMESTAMP* datatypes. You gain some storage space at the sacrifice of performance.

Use the following *DEFAULT*-related attributes to set various defaults:

[WITH] DEFAULT {CURRENT SCHEMA | USER | NULL | constant}

Declares a default for the column. That is, whenever an *INSERT* or *UPDATE* (or *LOAD*) statement does not provide a value for the column, then DB2 will use the default value you specify. *WITH* is a noise word. When omitted, *NULL* is always the default. The options for the *DEFAULT* clause are:

CURRENT SCHEMA

Inserts the *CURRENT SCHEMA* system value. The column must have a character datatype of sufficient length to store *CURRENT SCHEMA*.

USER

Inserts the *USER* system value. The column must have a character datatype of sufficient length to store *USER*.

NULL

Inserts a null value, the default.

constant

Can be any constant string, date, or number of 254 characters or fewer.

GENERATED {ALWAYS | BY DEFAULT}

Tells DB2 to generate values for the column, in particular for *IDENTITY* columns, either every time a record is inserted in the table (*ALWAYS*) or whenever a value is not provided for the column (*BY DEFAULT*).

AS (expression)

Assigns the default value based on the evaluation of an *expression*. The expression might be something like an external user-defined function, a subquery, or a stored procedure. Refer to the *ALTER COLUMN...generated expressions* material earlier in this section.

AS IDENTITY [START WITH int | INCREMENT BY int]

Declares that the column is an auto-incrementing identity for a column of a number datatype. Identities can be either ascending or descending in nature. DB2 allows only one identity column per table. All identity columns are implicitly *NOT NULL*. Identities are further defined by these optional characteristics:

START WITH int

Declares the seed value (i.e., the starting value), *int*, of the identity column. The default is *MINVALUE* for ascending identities and *MAXVALUE* for descending identities.

INCREMENT BY int

Declares the amount each entry to the identity will increase (if positive) or decrease (if negative). The default is 1. A negative *INCREMENT BY* value tells DB2 that the identity is descending in nature.

[NO] {MINVALUE | MAXVALUE} [int]

Declares the value, *int*, at which the identity either cycles to the beginning or stops generating values. Both clauses may appear as attributes of an identity. They are not mutually exclusive. *NO MINVALUE*, the default minimum value setting, tells DB2 to start ascending identities at the *START WITH* value or 1, if omitted. *MINVALUE int* tells DB2 the lowest allowable value for a descending identity. *NO MAXVALUE*, the default maximum value setting, tells DB2 to stop or recycle ascending identities at maximum value allowed for the datatype. For descending identities, the default is the *START WITH* value or -1, when omitted. *MINVALUE int* tells DB2 the highest allowable value for an ascending identity.

[NO] CYCLE

Declares that DB2 should recycle the identity values (*CYCLE*) when the maximum value is exceeded on ascending identities (or the minimum value is decremented on a descending identity) or that DB2 should stop generating identities (*NO CYCLE*) once the minimum or maximum is reached. *NO CYCLE* is the default. DB2 will generate duplicate values under the *CYCLE* setting.

[NO] CACHE [int]

Tells DB2 to cache some pre-allocated values in memory (*CACHE*) or not (*NO CACHE*). You may optionally tell DB2 how many identity values, *int*, to cache using *CACHE int*.

[NO] ORDER

Tells DB2 whether identity values must be generated in the order they were requested (*ORDER*) or not (*NOT ORDER*). When omitted, *NOT ORDER* is the default.

For example, our company has acquired another company with a whole roster of employees. We want to assign the new **employees** a new **emp_id** based on our current highest **emp_id** identity column value:

```
CREATE TABLE temp_emp
(new_emp_id INT NOT NULL GENERATED ALWAYS AS IDENTITY
 (START WITH 500, INCREMENT BY 1, NO CYCLE, CACHE),
dept    VARCHAR (50) NOT NULL,
mgr_id  INT,
location CHAR(30))
```



DB2 supports the creation of temporary tables, but not in the *CREATE TABLE* statement. Instead, use the *DECLARE GLOBAL TEMPORARY TABLE* statement described in the DB2 documentation.

However, be extra careful whenever altering a table. You should generally expect that any procedural objects that are dependent on the table, such as procedures or functions, will become invalid.

-- Creating a column-level constraint

```
CREATE TABLE favorite_books
(isbn    CHAR(100) PRIMARY KEY,
book_name VARCHAR(40) UNIQUE,
category VARCHAR(40) NULL,
subcategory VARCHAR(40) NULL,
pub_date DATETIME NOT NULL,
purchase_date DATETIME NOT NULL,
CONSTRAINT fk_categories FOREIGN KEY (category)
REFERENCES category(cat_name));
```

In the following example, we alter our example table, **favorite_books**, to not log SQL changes for replication. We also assign some free space to the data pages of the table:

```
ALTER TABLE favorite_books
PCTFREE 20
DATA CAPTURE NONE
```

We also finally decide that although the records in the table change quite frequently, the table structure itself should be static:

```
ALTER TABLE favorite_books NO VOLATILE CARDINALITY
```

MySQL

The MySQL syntax for *CREATE TABLE* creates a permanent or local temporary table within the database in which the

command is issued.

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name
{(column_name datatype attributes
  constraint_type constraint_name [,...])
 [constraint_type [constraint_name] [,...]]
 [MATCH FULL | MATCH PARTIAL]
 [ON {DELETE | UPDATE} {RESTRICT | CASCADE | SET NULL |
  NO ACTION | SET DEFAULT}]
LIKE other_table_name }
[[TYPE = {ISAM | MyISAM | HEAP | BDB | InnoDB | MERGE |
MRG_MyISAM} |
  AUTO_INCREMENT = int |
  AVG_ROW_LENGTH = int |
  CHECKSUM = {0 | 1} |
  COMMENT = "string" |
  DELAY_KEY_WRITE = {0 | 1} |
  MAX_ROWS = int |
  MIN_ROWS = int |
  PACK_KEYS = {0 | 1} |
  PASSWORD = "string" |
  ROW_FORMAT= { default | dynamic | static | compressed } |
  RAID_TYPE = {1 | STRIPED | RAID0} |
  RAID_CHUNKS = int |
  RAID_CHUNKSIZE = int |
  INSERT_METHOD = {NO | FIRST | LAST} |
  DATA DIRECTORY = "path to directory"
  INDEX DIRECTORY = "path to directory" }
[[IGNORE | REPLACE] select_statement]
```

The MySQL syntax for *ALTER TABLE* allows modifications to a table or even renaming of a table:

```
ALTER [IGNORE] TABLE table_name
{[ADD [COLUMN] (column_name datatype attributes )
 [FIRST | AFTER column_name]]
 | [ADD [UNIQUE | FULLTEXT] INDEX [index_name]
 | (index_col_name [,...]) ]
 | [ADD PRIMARY KEY (index_col_name [,...]) ]
 | [ALTER [COLUMN] column_name {SET DEFAULT literal |
 | DROP DEFAULT}]
 | [CHANGE | MODIFY] [COLUMN] old_col_name new_column_name]
```

[FIRST | AFTER *column_name*]
| [MODIFY] *column_name datatype attributes*
| [DROP [COLUMN] *column_name*]
| [DROP PRIMARY KEY]
| [DROP INDEX *index_name*]
| [{ENABLE | DISABLE} KEYS]
| [RENAME [TO] *new_tbl_name*]
| [table_options] } [,...]

Keywords and parameters are as follows:

TEMPORARY

Creates a table that persists for the duration of the connection under which it was created. Once that connection closes, the temporary table is automatically deleted.

IF NOT EXISTS

Prevents an error if the table already exists. A schema specification is not required.

constraint_type

Allows standard ANSI SQL constraints to be assigned at the column or table level. MySQL fully supports these constraints: primary key, unique, and default (must be a constant). MySQL provides syntax support for check, foreign key, and references constraints, but they are not functional except on InnoDB tables. MySQL also has three specialty constraints:

FULLTEXT [INDEX]

Creates a fulltext-search index to facilitate rapid searches of large blocks of text. Note that only MyISAM tables support *FULLTEXT* indexes and that they can only be made on *CHAR*, *VARCHAR*, and *TEXT* columns.

AUTO_INCREMENT

Sets up an integer column so that it automatically increases its value by 1 (starting with a value of 1). MySQL only allows one *AUTO_INCREMENT* column per table. When the maximum value is deleted, the value is reused. When all records are deleted (using *DELETE* or *TRUNCATE*), the values may start over. This option is allowed on ISAM tables, but not MyISAM and **InnoDB** tables.

[UNIQUE] INDEX

When an *INDEX* characteristic is assigned to a column, a name for the index also can be included. (A MySQL synonym for *INDEX* is *KEY*.) If a name is not assigned to the primary key, MySQL assigns a name of *column_name* plus a numeric suffix (*_2*, *_3*, . . .) to make it unique. All table types, except for the ISAM table type, support indexes on NULL columns, or on *BLOB* or *TEXT* datatype columns. Note that *UNIQUE* without *INDEX* is valid syntax on MySQL.

TYPE

Describe how the data should be physically stored. You can convert tables between types using the *ALTER TABLE* statement. Only InnoDB and BDB tables offer recoverability with the *COMMIT* or *ROLLBACK* statements. All other table types will result in the loss of data if the database crashes, but are much faster and take up less space. The default is MyISAM. Following is the list of MySQL table types:

BDB

Creates a transaction-safe table with page locking. Tables of this type are based on the BerkeleyDB transaction processing engine available at <http://www.sleepycat.com>. BDB maintains logfiles and keeps track of checkpoints and *ROLLBACK* and *COMMIT* statements. BDB tables must have a declared primary key. Some operations on BDB tables are slower than the same operations on MyISAM tables.

HEAP

Creates a memory resident table that uses hashed indexes. Since they are memory resident, they are not transaction safe. Any data that they contain will be lost if the server crashes. Think of HEAP tables as an alternative to temporary tables. If you use HEAP tables, always specify the *MAX_ROWS* option so that you do not use all available memory. *HEAP* tables also do not support *BLOB* columns, *TEXT* columns, *AUTO_INCREMENT* constraints, *ORDER BY* clauses, or variable-length record format.

InnoDB

Creates a transaction-safe table with row-level locking. It also provides an independently managed tablespace, checkpoints, non-locking reads and fast reads from large datafiles for heightened concurrency and performance. Requires the *innodb_data_file_path* startup parameter. InnoDB supports all ANSI constraints including *CHECK* and *FOREIGN KEY*. (InnoDB is a separate complete transaction processing engine available at <http://www.innodb.com>. Well-known developer web sites like Slashdot.org run on InnoDB.) InnoDB requires special setup, including a possible recompile of the MySQL kernel, *init* file parameters, and the creation of tablespaces. Although InnoDB is included by default in the binary distributions of MySQL 4.0 and greater, you should check with your DBA to ensure you can use InnoDB tables.

ISAM

Creates a table in the older ISAM style. ISAM is deprecated as a feature and will probably not be supported beyond MySQL v5.0.

MERGE | MRG_MyISAM

Collects several identically structured MyISAM tables for use as one table. The *MERGE* table type provides some of the performance benefits of a partitioned table. The two keywords, *MERGE* and *MRG_MyISAM*, are synonyms. *SELECT*, *DELETE*, and *UPDATE* statements operate against the collection of tables as if they were one table. Think of a merge table as the collection, but not as the table(s) containing the data. Dropping a merge table only removes it from the collection. It does not erase the table or data from the database. Specify a *MERGE* table by using the statement *UNION=(table1, table2, ...)*.

MyISAM

Stores data in *.MYD* files and indexes in *.MYI* files. MyISAM is based on ISAM code with several extensions. MyISAM is a binary storage structure that is more portable than ISAM. MyISAM supports *AUTO_INCREMENT* columns, compressed tables (with the *mysampack* utility), and large table sizes. Under MyISAM, *BLOB* and *TEXT* columns can be indexes, and up to 32 indexes are allowed per table.

AUTO_INCREMENT = *int*

Sets the auto_increment value, *int*, for the table (MyISAM only).

AVG_ROW_LENGTH = *int*

Sets an approximate average row length for tables with variable-size records. MySQL uses *avg_row_length* * *max_rows* to decide how big a table may be.

CHECKSUM = {0 | 1}

When set to 1, maintains a checksum for all rows in the table (MyISAM only). Makes processing slower, but is less prone to corruption.

COMMENT = "*string*"

Allows a comment of up to 60 characters.

DELAY_KEY_WRITE = {0 | 1}

When set to 1, delays key table updates until the table is closed (MyISAM only).

MAX_ROWS = *int*

Sets a maximum number of rows to store in the table. The default maximum is 4 GB of space.

MIN_ROWS = *int*

Sets a minimum number of rows to store in the table.

PACK_KEYS = {0 | 1}

When set to 1, compacts the indexes of the table, making reads faster but updates slower (MyISAM and ISAM only). By default, only strings are packed. When set to 1, both strings and numeric values are packed.

PASSWORD = "*string*"

Encrypts the *.FRM* file with a password, "*string*", but not the table itself.

ROW_FORMAT = { *default* | *dynamic* | *static* | *compressed* }

Determines how future rows should be stored in a MyISAM table. *DYNAMIC* allows the rows to be of variable sizes (i.e., using *VARCHAR*) while *FIXED* expects fixed size columns (i.e., *CHAR*, *INT*, etc.).

RAID_TYPE = {1 | *STRIPED* | *RAID0*}

Enables tables to cross the 2 gigabyte/4 gigabyte size limit for MyISAM datafiles (but not index files) on operating systems that don't support very large files. The *RAID_TYPE* clause is intended as a workaround for database servers experiencing the aforementioned size limits, so you should not use *RAID_TYPE* on operating systems that support larger files. The keywords 1, *STRIPED*, and *RAIDZERO* are all synonyms, where *STRIPED* is the most commonly seen. The *--with-raid* init parameter is required. Currently, MySQL handles the *RAID_CHUNKS* and *RAID_CHUNKSIZE* clauses automatically.

RAID_CHUNKS = *int*

Overrides the default subdirectories used for the *RAID_TYPE*. MySQL usually creates subdirectories named 00, 01, and 02 in the database directory for all each MyISAM table. Each subdirectory will contain a file called *my_table_name.MYD*.

RAID_CHUNKSIZE = *int*

Sets a chunksize other than the default of 1,024 bytes. As the file size for the MyISAM table grows, MySQL will allocate additional space in *int* bytes in a round-robin fashion.

INSERT_METHOD = {*NO* | *FIRST* | *LAST*}

Required for *MERGE* tables. If no setting is specified for a *MERGE* table or the value is *NO*, then *INSERTS* are disallowed. *FIRST* inserts all records to the first table in the collection, while *LAST* inserts them all to the last table of the collection

DATA DIRECTORY = "*path to directory*"

Overrides the default path and directory that MySQL should use for MyISAM table storage.

INDEX DIRECTORY = "*path to directory*"

Overrides the default path and directory that MySQL should use for MyISAM index storage.

[IGNORE | REPLACE] *SELECT_statement*

Creates a table with columns based upon the elements listed in the *SELECT* statement. The new table will be populated with the results of the *SELECT* statement if the statement returns a result set.

ALTER [IGNORE]

The altered table will include all duplicate records unless the *IGNORE* keyword is used. Otherwise, the statement will fail if a duplicate row is encountered, the table has a unique index or primary key, and *IGNORE* is not used.

{ADD | COLUMN} [FIRST | AFTER *column_name*]

Adds or moves a column, index, or key to the table. When adding or moving columns, the new column appears as the last column on the table unless it is placed *AFTER* another named column.

ALTER COLUMN

Allows the definition or reset of a default value for a column. If you reset a default, MySQL will assign a new default value to the column.

CHANGE

Renames a column or changes its datatype.

MODIFY

Changes a column's datatype or attributes such as *NOT NULL*. Existing data in the column is automatically converted to the new datatype.

DROP

Drops the column, key, or index. A dropped column is also removed from any indexes it participated in. When dropping a primary key, MySQL will drop the first unique key if no primary key is present.

{ENABLE | DISABLE} KEYS

Enables or disables all non-unique keys on a MyISAM table simultaneously. This can be useful for bulk loads where you want to temporarily disable constraints until after the load is finished. It also speeds performance by finishing all index block flushing at the end of the operation.

RENAME [TO] *new_tbl_name*

Renames a table.

For example:

```
CREATE TABLE test_example
(column_a INT NOT NULL AUTO_INCREMENT,
PRIMARY KEY(column_a),
INDEX(column_b))
TYPE=HEAP
IGNORE
SELECT column_b,column_c FROM samples;
```

This creates a heap table with three columns: **column_a**, **column_b**, and **column_c**. Later on, we could change this

table to the MyISAM type:

```
ALTER TABLE example TYPE=MyISAM;
```

Three operating-system files are created when MySQL creates a MyISAM table: a table definition file with the extension *.FRM*, a datafile with the extension *.MYD*, and an index file with the extension *.MYI*. The *.FRM* datafile is used for all other tables.

The following example creates two base MyISAM tables and then creates a *MERGE* table from them:

```
CREATE TABLE message1
  (message_id INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20));
CREATE TABLE message2
  (message_id INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20));
CREATE TABLE all_messages
  (message_id INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20))
  TYPE=MERGE UNION=(message1, message2) INSERT_METHOD=LAST;
```

The following example renames both a table and a column:

```
ALTER TABLE employee RENAME AS emp;
ALTER TABLE employee CHANGE employee_ssn emp_ssn INTEGER;
```

Since MySQL allows the creation of indexes on a portion of a column (for example, on the first ten characters of a column), you can also build short indexes on very large columns.

MySQL can redefine the datatype of an existing column. To avoid losing any data, the values contained in the column must be compatible with the new datatype. For example, a date column could be redefined to a character datatype, but not a character datatype to an integer. Here's an example:

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

MySQL allows some additional flexibility in the *ALTER TABLE* statement by allowing users to issue multiple *ADD*, *ALTER*, *DROP*, and *CHANGE* clauses in a comma-delimited list in a single *ALTER TABLE* statement. However, be aware that the *CHANGE column_name* and *DROP INDEX* clauses are MySQL extensions not found in SQL2003. MySQL supports the clause *MODIFY column_name* to mimic the same feature found in Oracle.

Oracle

The Oracle syntax for *CREATE TABLE* creates a relational table either by declaring the structure or by referencing an existing table. You can modify a table after it is created using the *ALTER TABLE* statement. Oracle also allows the creation of a relational table that uses user-defined types for column definitions. You can also create an *object table* that is explicitly defined to hold a specific UDT, usually a *VARRAY* or *NESTED TABLE* type, as well as XMLType tables.

The standard ANSI style *CREATE TABLE* statement is supported, but Oracle has added many sophisticated extensions to the command. For example, Oracle allows significant control over the storage and performance parameters of a table. In both the *CREATE TABLE* and *ALTER TABLE* statements, you'll see a great deal of nesting and reusable clauses. To make this somewhat easier to read, we have broken Oracle's *CREATE TABLE* statement into three distinct variations (relational table, object table, and XML table) so that you can more easily follow the syntax.

The *CREATE TABLE* syntax for a standard relational table, which has no object or XML properties, is as follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
  [( {column | attribute} [SORT] [DEFAULT expression] [{column_constraint |
```

```
    inline_ref_constraint} ] |
    {table_constraint_clause | table_ref_constraint} |
    {GROUP log_group (column [NO LOG] [,...]) [ALWAYS] | DATA
      (constraints [,...]) COLUMNS} ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[table_constraint_clause]
{ [physical_attributes_clause] [TABLESPACE tablespace_name]
  [storage_clause] [[NO]LOGGING] |
  [CLUSTER (column [,...]) ] |
  {[ORGANIZATION
    {HEAP [physical_attributes_clause][TABLESPACE
      tablespace_name][storage_clause]
      [COMPRESS | NOCOMPRESS] [[NO]LOGGING] |
    INDEX [physical_attributes_clause][TABLESPACE
      tablespace_name][storage_clause]
      [PCTTHRESHOLD int] [COMPRESS [int] | NOCOMPRESS]
      [MAPPING TABLE | NOMAPPING][...] [[NO]LOGGING]
      [[INCLUDING column] OVERFLOW
        [physical_attributes_clause][TABLESPACE tablespace_name]
        [storage_clause] [[NO]LOGGING] } } ]
    EXTERNAL ( [TYPE driver_type] ) DEFAULT DIRECTORY directory_name
      [ACCESS PARAMETERS {USING CLOB subquery | ( opaque_format ) } ]
      LOCATION ( [directory_name:]'location_spec' [,...])
      [REJECT LIMIT {int | UNLIMITED} } } }
  {[ENABLE | DISABLE} ROW MOVEMENT]
  [[NO]CACHE] [[NO]MONITORING] [[NO]ROWDEPENDENCIES]
  [PARALLEL int | NOPARALLEL] [NOSORT] [[NO]LOGGING]
  [COMPRESS [int] | NOCOMPRESS]
  {[ENABLE | DISABLE} [[NO]VALIDATE]
    {UNIQUE (column [,...]) | PRIMARY KEY | CONSTRAINT constraint_name} ]
    [USING INDEX {index_name | CREATE_INDEX_statement} ] [EXCEPTIONS INTO]
    [CASCADE] [{KEEP | DROP} INDEX] ] |
  [partition_clause]
  [AS subquery]
```

The relational table syntax contains a large number of optional clauses. However, the table definition must contain, at a minimum, either column names and datatypes specifications or the AS *subquery* clause.

The Oracle syntax for an object table follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
```

```
AS object_type [[NOT] SUBSTITUTABLE AT ALL LEVELS]
[ ( { column | attribute } [DEFAULT expression] [{ column_constraint |
    inline_ref_constraint } ] |
  { table_constraint_clause | table_ref_constraint } |
  { GROUP log_group (column [NO LOG] [,...]) [ALWAYS] | DATA
    (constraints [,...]) COLUMNS } ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX [index_name] ([physical_attributes_clause] [storage_clause]) ]
[physical_attributes_clause] [TABLESPACE tablespace_name] [storage_clause]
```

The Oracle syntax for an XMLType table follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
OF XMLTYPE
[ ( { column | attribute } [DEFAULT expression] [{ column_constraint |
    inline_ref_constraint } ] |
  { table_constraint_clause | table_ref_constraint } |
  { GROUP log_group (column [NO LOG] [,...]) [ALWAYS] | DATA
    (constraints [,...]) COLUMNS } ) ]
[XMLTYPE {OBJECT RELATIONAL [xml_storage_clause] |
  CLOB [lob_segname] [lob_params]}] [xml_schema_spec]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX index_name ([physical_attributes_clause] [storage_clause]) ]
[physical_attributes_clause] [TABLESPACE tablespace_name] [storage_clause]
```

The Oracle syntax for *ALTER TABLE* changes the table or column properties, storage characteristics, *LOB* or *VARRAY* properties, partitioning characteristics, and integrity constraints associated with a table and/or its columns. The statement does other things like move an existing table into a different tablespace, free recuperated space, compact the table segment, and adjust the "high-water mark."

The ANSI SQL standard uses the *ALTER* keyword to modify existing elements of a table, while Oracle uses *MODIFY* to mean the same thing. Since they are essentially the same thing, please consider the behavior of otherwise identical clauses (for example, ANSI's *ALTER TABLE...ALTER COLUMN* to Oracle's *ALTER TABLE...MODIFY COLUMN*) to be functionally equivalent.

Oracle's *ALTER TABLE* syntax is:

```
ALTER TABLE table_name
-- alter table characteristics
[physical_attributes_clause] [storage_clause]
[[NO]LOGGING] [[NO]CACHE] [[NO]MONITORING] [[NO]COMPRESS]
[SHRINK SPACE [COMPACT] [CASCADE] ]
[UPGRADE [[NOT] INCLUDING DATA] column_name datatype attributes]
```

```
[[NO]MINIMIZE RECORDS_PER_BLOCK]

[PARALLEL int | NOPARALLEL]

[{ENABLE | DISABLE} ROW MOVEMENT]

[{ADD | DROP} SUPPLEMENTAL LOG

  {GROUP log_group [(column_name [NO LOG] [...]) [ALWAYS] ] |
  DATA ( {ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY} [...] ) COLUMNS} ]

[ALLOCATE EXTENT

  [( SIZE int[K|M|G|T] ) [DATATILE 'filename' ] [INSTANCE int] ) ] ]

[DEALLOCATE UNUSED [KEEP int[K|M|G|T] ] ]

[ORGANIZATION INDEX ...

  [COALESCE] [MAPPING TABLE | NOMAPPING] [{PCTTHRESHOLD int]

  [COMPRESS int | NOCOMPRESS]

  [ { ADD OVERFLOW [TABLESPACE tablespace_name] [[NO]LOGGING

    [physical_attributes_clause] ] |

  OVERFLOW { [ALLOCATE EXTENT (SIZE int [K | M]) [DATAFILE

    'filename' ] [INSTANCE int] ) |

    [DEALLOCATE UNUSED [KEEP int [K | M]] } } ] ]

[RENAME TO new_table_name]

-- alter column characteristics

[ADD (column_name datatype attributes [...] ) ]

[DROP { {UNUSED COLUMNS | COLUMNS CONTINUE} [CHECKPOINT int] |

  {COLUMN column_name | (column_name [...]) } [CHECKPOINT int]

  [{CASCADE CONSTRAINTS | INVALIDATE}] } ]

[SET UNUSED {COLUMN column_name | (column_name [...]) }

  [{CASCADE CONSTRAINTS | INVALIDATE}]]

[MODIFY { (column_name datatype attributes [...] ) |

  COLUMN column_name [NOT] SUBSTITUTABLE AT ALL LEVELS [FORCE] } ]

[RENAME COLUMN old_column_name TO new_column_name]

[MODIFY {NESTED TABLE | VARRAY} collection_item [RETURN AS {LOCATOR |

  VALUE}]]

-- alter constraint characteristics

[ADD CONSTRAINT constraint_name table_constraint_clause]

[MODIFY CONSTRAINT constraint_name constraint_state_clause]

[RENAME CONSTRAINT old_constraint_name TO new_constraint_name]

[DROP { { PRIMARY KEY | UNIQUE (column [...]) } [CASCADE]

  [{KEEP | DROP} INDEX] |

  CONSTRAINT constraint_name [CASCADE] } ]

-- alter table partition characteristics

[alter partition clauses]
```

```
-- alter external table characteristics

DEFAULT DIRECTORY directory_name

    [ACCESS PARAMETERS {USING CLOB subquery | ( opaque_format ) } ]

    LOCATION ( [ directory_name: ]'location_spec' [,...] )

[ADD ( column_name ...)][DROP column_name ...][MODIFY ( column_name ...)]

[PARALLEL int | NOPARALLEL]

[REJECT LIMIT {int | UNLIMITED} ]

[PROJECT COLUMN {ALL | REFERENCED} ] } }

-- move table clauses

[MOVE [ONLINE] [ physical_attributes_clause ]

    [TABLESPACE tablespace_name] [[NO]LOGGING] [PCTTHRESHOLD int]

    [COMPRESS int | NOCOMPRESS] [MAPPING TABLE | NOMAPPING]

    [[INCLUDING column] OVERFLOW

    [ physical_attributes_clause ] [TABLESPACE tablespace_name]

    [[NO]LOGGING]] ]

[LOB ...] [VARRAY ...] [PARALLEL int | NOPARALLEL ]

-- enable/disable attributes and constraints

[ { {ENABLE | DISABLE} [[NO]VALIDATE]} {UNIQUE ( column [,...] ) |

    PRIMARY KEY | CONSTRAINT constraint_name} ]

[USING INDEX { index_name | CREATE_INDEX_statement |

    [TABLESPACE tablespace_name] [ physical_attributes_clause ]

    [storage_clause]

    [NOSORT] [[NO]LOGGING] [ONLINE] [COMPUTE STATISTICS]

    [COMPRESS | NOCOMPRESS] [REVERSE]

    [ {LOCAL | GLOBAL} partition_clause ]

    [EXCEPTIONS INTO table_name] [CASCADE] [ {KEEP | DROP} INDEX ] ] |

    [ {ENABLE | DISABLE} ] [ {TABLE LOCK | ALL TRIGGERS} } ] }
```

The parameters are as follows:

column_constraint

Specifies a column constraint using the syntax found in the description later.

GROUP *log_group* (*column* [NO LOG] [,...]) [ALWAYS] | DATA (*constraints* [,...]) COLUMNS

Specifies a log group rather than a single logfile for the table.

ON COMMIT {DELETE | PRESERVE} ROWS

Declares whether a declared temporary table should keep the data in the table active for the entire session (PRESERVE) or only for the transaction that the temporary table is created in (DELETE).

table_constraint_clause

Specifies a table constraint using the syntax found in the description later.

physical_attributes_clause

Specifies the physical attributes of the table using the syntax found in the description later.

TABLESPACE *tablespace_name*

Specifies the name of the tablespace where the table you are creating will be stored. If omitted, the default tablespace for the schema owner will be used. See below for specifics. See the Oracle Concepts manual to learn about tablespaces and their use.

storage_clause

Specifies physical storage characteristics of the table using the syntax found in the description later.

[NO]LOGGING

Specifies that redo log records will be written during object creation (*LOGGING*) or not (*NOLOGGING*). *LOGGING* is the default. In case of database failure under the *NOLOGGING* option, the operation cannot be recovered by applying logfiles, and the object must be recreated. *NOLOGGING* can speed the creation of database objects. The *LOGGING* clause replaces the older *RECOVERABLE* clause, which is deprecated.

CLUSTER (*column* [,...])

Declares that the table is part of a clustered index. The *column* list should correspond, one to one, with the columns in a previously declared clustered index. Because it uses the clustered indexes space allocation, the *CLUSTER* clause is compatible with the *physical_attributes_clause*, *storage_clause*, or *TABLESPACE* clause. Tables containing *LOBs* are incompatible with the *CLUSTER* clause.

ORGANIZATION HEAP

Declares how the data of the table should be recorded to disk. *HEAP*, the default for an Oracle table, declares that no order is associated with the storage of rows of data (i.e., the physical order in which records are written to disk) for this table. The *ORGANIZATION HEAP* clause allows several optional clauses, described in detail elsewhere in this list, control storage, logging, and compression for the table.

ORGANIZATION INDEX

Declares how the data of the table should be recorded to disk. *INDEX* declares that the records of the table are physically written to disk in the sort order defined by the primary key of the table. Oracle calls this an index-organized table. A primary key is required. Note that the *physical_attributes_clause*, the *TABLESPACE* clause, and the *storage_clause* — all described in greater detail elsewhere in this section — as well as the *[NO]LOGGING* keyword may all be associated with the new *INDEX* segment as you create it. In addition, the following subclauses may also be associated with an *ORGANIZATION INDEX* clause:

PCTTHRESHOLD *int*

Declares the percentage, *int*, of space in each index block preserved for data. Data, on a record by record basis, that cannot fit in this space will be placed in the overflow segment.

INCLUDING *column*

Declares the point at which a record is to split between index and overflow portions. All columns that follow the specified *column* will be stored in the overflow segment. The *column* cannot be a primary key column.

MAPPING TABLE | NOMAPPING

Tells the database to create a mapping of local to physical ROWIDs. This mapping is required to create a bitmap index on an IOT. Mappings are also partitioned identically if the table is partitioned. *NOMAPPING* tells the database not to create the ROWID map.

[INCLUDING *column*] OVERFLOW

Declares that a record that exceeds the *PCTTHRESHOLD* value be placed in a segment described in this clause. The *physical_attributes_clause*, the *TABLESPACE* clause, the *storage_clause* — all described elsewhere in the list in greater detail — and the *[NO]LOGGING* keyword may all be associated with a specific *OVERFLOW* segment when you create it. The optional *INCLUDING column* clause defines a column at which to divide an IOT row into index and overflow portions. Primary key columns are always stored in the index. However, all non-primary key columns that follow *column* are stored in the overflow data segment.

ORGANIZATION EXTERNAL

Declares how the data of the table should be recorded to disk. *EXTERNAL* declares that the table stores its data outside of the database and is usually read-only. Its metadata is stored in the database but its data is stored outside of the database. There are some restrictions on external tables. They cannot be temporary. They cannot have constraints. They can only have column, datatype, and attribute column-descriptors. *LOB* and *LONG* datatypes are disallowed. No other *ORGANIZATION* clauses are allowed with *EXTERNAL*. The following subclauses may be used with the *ORGANIZATION EXTERNAL* clause:

TYPE *driver_type*

Defines the access driver API for the external table. The default is *ORACLE_LOADER*.

DEFAULT DIRECTORY *directory_name*

Defines the default directory on the filesystem where the external table resides.

ACCESS PARAMETERS {USING CLOB *subquery* | (*opaque_format*) }

Assigns and passes specific parameters to the access driver. Oracle does not interpret this information. *USING CLOB subquery* tells Oracle to derive the parameters and their values from a subquery that returns a single row with a single column of datatype *CLOB*. The subquery cannot contain an *ORDER BY*, *UNION*, *INTERSECT*, or *MINUS/EXCEPT*. The *opaque_format* clause allows you to list parameters and their values as described in the ORACLE LOADER section of the "Oracle9i Database Utilities" guide.

LOCATION (*directory_name*:'*location_spec*' [,...])

Defines one or more external data sources, usually as files. Oracle does not interpret this information.

REJECT LIMIT {*int* | UNLIMITED}

Defines the number of conversion errors, *int*, that are allowed during the query to the external data source before Oracle aborts the query and returns an error. *UNLIMITED* tells Oracle to continue with the query no matter how many errors are encountered. The default is 0.

{*ENABLE* | *DISABLE*} ROW MOVEMENT

Specifies that a row may be moved to a different partition or subpartition if required due to an update of the key (*ENABLE*) or not (*DISABLE*). The *DISABLE* keyword also specifies that Oracle return an error if an update to a key would require a move.

[NO]CACHE

Buffers a table for rapid reads, while *NOCACHE* turns off this behavior. Index-organized tables offer *CACHE* behavior.

[NO]MONITORING

Specifies that modification statistics can be collected for this table (*MONITORING*) or not (*NOMONITORING*). *NOMONITORING* is the default.

[NO]ROWDEPENDENCIES

Specifies whether a table will use *row-level dependency tracking*, a feature that applies a system change number (SCN) greater than or equal to the last time of the transaction affecting the row. The SCN adds 6 extra bytes of space to each record. Row-level dependency tracking is most useful in replicated environments with parallel data propagation. *NOROWDEPENDENCIES* is the default.

PARALLEL *int* | NOPARALLEL

Allows for the parallel creation of the table by distinct CPUs to speed the operation. It also enables parallelism for queries and other data-manipulation operations against the table after its creation. An optional integer value may be supplied to define the exact number of parallel threads used to create the table in parallel, as well as the parallel threads allowed to service the table in the future. (Oracle calculates the best number of threads to use in a given parallel operation, so the *PARALLEL* clause is an optional feature.) *NOPARALLEL*, the default, creates the table serially and disallows future parallel queries and data-manipulation operations.

COMPRESS [*int*] | NOCOMPRESS

Specifies if the table should be compressed or not. On index-organized tables, only the key is compressed; on heap-organized tables, the entire table is compressed. This can greatly reduce the amount of space consumed by the table. *NOCOMPRESS* is the default. In index-organized tables, you can specify the number of prefix columns, *int*, to compress. The default value for *int* is the number of keys in the primary key minus one. You need not specify an *int* value on other clauses such as *ORGANIZATION*. When you omit the *int* value, Oracle will apply compression to the entire table.

{ENABLE | DISABLE} [[NO]VALIDATE]] {UNIQUE (*column* [,...]) | PRIMARY KEY | CONSTRAINT *constraint_name*}

Declares whether the named key or constraint applies to all of the data in the new table. *ENABLE* specifies that the key or constraint applies to all new data in the table. *DISABLE* specifies that the key or constraint is disabled for the new table, with the following options:

[NO]VALIDATE

Verifies that all existing data in the table complies with the key or constraint (*VALIDATE*) or not (*NOVALIDATE*). When *NOVALIDATE* is specified with *ENABLE*, Oracle does not verify that existing data in the table complies with the key or constraint, but ensures that new data added to the table does comply with the constraint.

{UNIQUE (*column* [,...]) | PRIMARY KEY | CONSTRAINT *constraint_name*}

Declares the unique constraint, primary key, or constraint that is enabled or disabled.

USING INDEX *index_name* | *CREATE_INDEX_statement*

Declares the name, *index_name*, of a pre-existing index (and its characteristics) used to enforce the key or constraint, or creates a new index, *CREATE_INDEX_statement*. If neither are declared, Oracle creates a new index.

EXCEPTIONS INTO

Specifies the name of a table into which Oracle places information about rows violating the constraint. Run the *utlexpt1.sql* script before using this keyword to explicitly create this table.

CASCADE

Cascades the disablement/enabling to any integrity constraints that depend on the constraint named in the clause. Usable only with the *DISABLE* clause.

{KEEP | DROP} INDEX

Lets you keep (*KEEP*) or drop (*DROP*) an index used to enforce a unique or primary key. You can drop the key only when disabling it.

partition_clause

Declares partitioning and subpartitioning of a table. Partitioning syntax can be quite complex. Refer to the material later in this section under [Oracle partitioned and subpartitioned tables](#) for the full syntax and examples.

AS subquery

Declares a *subquery* that inserts rows into the table upon creation. The column names and datatypes used in the subquery can act as substitutes for column name and attribute declarations for the table.

AS object_type

Declares that the table is based on a pre-existing object type.

[NOT] SUBSTITUTABLE AT ALL LEVELS

Declares whether row objects corresponding to subtypes can be inserted into the type table or not. When this clause is omitted, the default is *SUBSTITUTABLE AT ALL LEVELS*.

inline_ref_constraint and *table_ref_constraint*

Declares a reference constraint used by an object-type table or XMLType table. These clauses are described in greater detail later in this section under [Oracle XMLType and object-type tables](#).

OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}

Declares whether the object ID (OID) of the object-type table is *SYSTEM GENERATED* or based on the *PRIMARY KEY*. When omitted, the default is *SYSTEM GENERATED*.

OIDINDEX [index_name]

Declares an index, and possibly a name for the index, if the OID is system generated. You may optionally apply a *physical_attributes_clause* and a *storage_clause* to the *OIDINDEX*. If the OID is based on the primary key, then this clause is unnecessary.

OF XMLTYPE

Declares that the table is based on Oracle's *XMLTYPE* datatype.

XMLTYPE {OBJECT RELATIONAL [xml_storage_clause] | CLOB [lob_segname] [lob_params]}

Declares how the underlying storage of the *XMLTYPE* is handled. *OBJECT RELATIONAL* stores the data in object-relational columns. This subclause requires an *xml_schema_spec* and a schema that has been preregistered using the *DBMS_XMLSCHEMA* package. *CLOB* specifies that the *XMLTYPE* data will be stored in a *LOB* column. You may optionally specify the *LOB* segment name and/or the *LOB* storage parameters.

xml_schema_spec

Allows you to specify the URL of a registered XML schema and an XML element name. The element name is required but the URL is optional.

ADD ...

Adds a new column, constraint, overflow segment, or supplemental log group to an existing table.

MODIFY ...

Changes an existing column, constraint, or supplemental log group on an existing table.

DROP ...

Drops an existing column, constraint, or supplemental log group from an existing table. You can explicitly drop columns marked as unused from a table with *DROP UNUSED COLUMNS*; however, Oracle will also drop all unused columns when *any* other column is dropped. The *INVALIDATE* keyword causes any object that depends on the dropped object to become invalid and unusable until it is recompiled or when it is next used. The *COLUMNS CONTINUE* clause is used only when a *DROP COLUMN* statement failed with an error and you wish to continue where it left off.

RENAME ...

Renames an existing table, column, or constraint on an existing table.

SET UNUSED ...

Declares a column or columns to be unused. The column or columns are no longer accessible from *SELECT* statements, though they still count toward the maximum number of columns allowed per table (1000). *SET UNUSED* is the fastest way to render a column unusable within a table, but it is not the best way. Only use *SET UNUSED* as a shortcut until you can actually use *ALTER TABLE...DROP* to drop the column.

COALESCE

Merges the contents of index blocks used to maintain the index-organized table so that the blocks can be reused. *COALESCE* is similar to *SHRINK*, though *COALESCE* compacts the segments less densely than does *SHRINK* and it does not release unused space.

ALLOCATE EXTENT

Explicitly allocates a new extent for the table using the specified parameters. You may mix and match any of the parameters. The size of the extents may be in bytes (no suffix), kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T).

DEALLOCATE UNUSED [KEEP int[K | M | G | T]]

Deallocates unused space at the end of the table, *LOB* segment, partition, or subpartition. The deallocated space is then usable by other objects in the database. The *KEEP* keyword tells how much space you want to have left over after deallocation is complete.

SHRINK SPACE [COMPACT] [CASCADE]

Shrinks the table, index-organized table, index, partition, subpartition, materialized view, or materialized log view, though only segments in tablespaces with automatic segment management may be shrunk. Shrinking a segment moves rows in the table, so make sure *ENABLE ROW MOVEMENT* is also used in the *ALTER TABLE...SHRINK* statement. Oracle compacts the segment, releases the emptied space, and adjusts the high-water mark unless the optional keywords *COMPACT* and/or *CASCADE* are applied. The *COMPACT* keyword only defragments the segment space and compacts the table row for subsequent release, but it does not readjust the high-water mark nor empty the space immediately. The *CASCADE* keyword performs the same shrinking operation (with some restrictions and exceptions) on all dependent objects of the table, including secondary indexes on index-organized tables. Used only with *ALTER TABLE*.

UPGRADE [NOT] INCLUDING DATA

Converts the metadata of object tables and relational tables with object columns to the latest version for each referenced type. The *INCLUDING DATA* clause will convert the data to the latest type format (*INCLUDING DATA*) or leave it unchanged (*NOT INCLUDING DATA*).

MOVE ...

Moves the tablespace, index-organized table, partition, or subpartition to a new location on the filesystem.

[NO]MINIMIZE RECORDS_PER_BLOCK

Tells Oracle to restrict or leave open the number of records allowed per block. The *MINIMIZE* keyword tells Oracle to calculate the maximum number of records per block and set the limit at that number. (Best to do this when a representative amount of data is already in the table.) This clause is incompatible with nested tables and index-organized tables. *NOMINIMIZE* is the default.

PROJECT COLUMN {REFERENCE | ALL}

Determines how the driver for the external data source validates the rows of the external table in subsequent queries. *REFERENCE* processes only those columns in the select item list. *ALL* processes the values in all columns, even those not in the select item list, and validates rows with full and valid column entries. Under *ALL*, rows are rejected when errors occur, even on columns that are not selected. *ALL* returns consistent results, while *REFERENCE* returns varying numbers of rows depending on the columns referenced.

{ENABLE | DISABLE} {TABLE LOCK | ALL TRIGGERS}

Enables or disables table-level locks and all triggers on the table, respectively. *ENABLE TABLE LOCK* is required if you wish to change the structure against an existing table, but it is not required when changing or reading the data of the table.

A global temporary table is available to all user sessions, but the data stored within a global temporary table is visible only to the session that inserted it. The *ON COMMIT* clause, which is allowed only when creating temporary tables, tells Oracle to either truncate the table after each commit against the table (*DELETE ROWS*) or to truncate the table when the session terminates (*PRESERVE ROWS*). For example:

```
CREATE GLOBAL TEMPORARY TABLE shipping_schedule
(
  ship_date DATE,
  receipt_date DATE,
  received_by VARCHAR2(30),
  amt NUMBER)
ON COMMIT PRESERVE ROWS;
```

The *CREATE TABLE* statement shown above creates a global temporary table, **shipping_schedule**, which retains inserted rows across multiple transactions.

The Oracle *physical_attributes_clause*

The *physical_attributes_clause* (as shown in the following code block) defines storage characteristics for an entire local table, or if the table is partitioned, for a specific partition (discussed later). To declare the physical attributes of a new table or change the attributes on an existing table, simply declare the new values:

```
-- physical_attributes_clause
[{PCTFREE int | PCTUSED int | INITRANS int |
MAXTRANS int | storage_clause}]
```

where:

PCTFREE int

Defines the percentage of free space reserved for each data block in the table. For example, a value of 10 reserves 10% of the data space for new rows to be inserted.

PCTUSED int

Defines the minimum percentage of space allowed in a block before it can receive new rows. For example, a value of 90 means new rows are inserted in the data block when the space used falls below 90%. The sum of *PCTFREE* and *PCTUSED* cannot exceed 100.

INITRANS int

Rarely tinkered with; it defines the allocation of from 1 to 255 initial transactions to a data block.

MAXTRANS int

Defines the maximum number of concurrent transactions on a data block.

The Oracle storage_clause and LOBs

The *storage_clause* controls a number of attributes governing the physical storage of data:

-- *storage_clause*

```
STORAGE ( [ {INITIAL int [K | M]
           | NEXT int [K | M]
           | MINEXTENTS int
           | MAXEXTENTS { int | UNLIMITED}
           | PCTINCREASE int
           | FREELISTS int
           | FREELIST GROUPS int
           | BUFFER_POOL {KEEP | RECYCLE | DEFAULT} ] [...] )
```

When delineating the storage clause attributes, enclose them in parentheses and separate them with spaces—for example, (*INITIAL 32M NEXT8M*). The attributes are as follows:

INITIAL int [K | M]

Sets the initial extent size of the table in bytes, kilobytes (K), or megabytes (M). Oracle 10g allows [K|M|G|T] to describe the size allocation in bytes, kilobytes (K), megabytes (M), gigabytes (G), and terabytes (T).

NEXT int [K | M]

Tells how much additional space to allocate after *INITIAL* is filled.

MINEXTENTS int

Tells Oracle to create a minimum number of extents. By default, only one is created, but more can be created when the object is initialized.

MAXEXTENTS int

Tells Oracle the maximum extents allowed. This value may be set to *UNLIMITED*. (Note that *UNLIMITED* should be used with caution since a table could grow until it consumes all free space on a disk.)

PCTINCREASE int

Controls the growth rate of the object after the first growth. The initial extent gets allocated as specified. The second extent is the size specified by *NEXT*. The third extent is *NEXT + (NEXT * PCTINCREASE)*. When *PCTINCREASE* is set to 0, *NEXT* is always used. Otherwise, each added extent of storage space is *PCTINCREASE* larger than the previous extent.

FREELISTS int

Establishes the number of freelists for each group, defaulting to 1.

FREELIST GROUPS int

Sets the number of groups of freelists, defaulting to 1.

BUFFER_POOL {KEEP | RECYCLE | DEFAULT}

Specifies a default buffer pool or cache for any non-cluster table where all object blocks are stored. Index-organized tables may have a separate buffer pool for the index and overflow segments. Partitioned tables inherit the buffer pool from the table definition unless they are specifically assigned a separate buffer pool.

KEEP

Puts object blocks into the *KEEP* buffer pool; that is, directly into memory. This enhances performance by reducing I/O operations on the table. *KEEP* takes precedence over the *NOCACHE* clause.

RECYCLE

Puts object blocks into the *RECYCLE* buffer pool.

DEFAULT

Puts object blocks into the *DEFAULT* buffer pool. When omitted, *DEFAULT* is the default buffer pool behavior.

For example, the table **books_sales** is defined on the **sales** tablespace as consuming an initial 8 MB of space, to grow by no less than 8 MB when the first extent is full. The table has no less than 1 and no more than 8 extents, limiting its maximum size to 64 MB:

```
CREATE TABLE book_sales
  (qty NUMBER,
   period_end_date DATE,
   period_nbr NUMBER)
TABLESPACE sales
STORAGE (INITIAL 8M NEXT 8M MINEXTENTS 1 MAXEXTENTS 8);
```

An example of a *LOB* table called **large_objects** with special handling for text and image storage might look like:

```
CREATE TABLE large_objects
  (pretty_picture BLOB,
   interesting_text CLOB)
STORAGE (INITIAL 256M NEXT 256M)
LOB (pretty_picture, interesting_text)
  STORE AS (TABLESPACE large_object_segment
           STORAGE (INITIAL 512M NEXT 512M)
           NOCACHE LOGGING);
```

The exact syntax used to define a *LOB*, *CLOB*, or *NLOB* column is defined by the *lob_parameter_clause*. *LOBs* can appear at many different levels within an Oracle table. For instance, separate *LOB* definitions could exist in a single table in a partition definition, a subpartition definition, and at the top table-level. The syntax of *lob_parameter_clause* follows:

```
{TABLESPACE tablespace_name} [{ENABLE | DISABLE} STORAGE IN ROW]
[storage_clause] [CHUNK int] [PCTVERSION int]
[RETENTION] [FREEPOOLS int]
```

[[CACHE | {NOCACHE | CACHE READS} [{LOGGING | NOLOGGING}]] }

In the *lob_parameter_clause*, each parameter is identical to those of the wider *CREATE TABLE LOB*-object level. However, the following four parameters are unique to *LOBs*:

{ENABLE | DISABLE} STORAGE IN ROW

Defines whether the *LOB* value is stored inline with the other columns of the row and the *LOB* locator (*ENABLE*), when it is smaller than approximately 4,000 bytes or less, or outside of the row (*DISABLE*). This setting cannot be changed once it is set.

CHUNK *int*

Allocates *int* number of bytes for *LOB* manipulation. *int* should be a multiple of the database blocksize, otherwise Oracle will round up. *int* should also be less than or equal to the value of *NEXT*, from the *storage_clause*, or else an error results. When omitted, the default chunk size is one block. This setting cannot be changed once it is set.

PCTVERSION *int*

Defines the maximum percentage, as *int*, of the overall *LOB* storage dedicated to maintaining old versions of the *LOB*. When omitted, the default is 10%.

RETENTION

Used in place of *PCTVERSION* on databases in automatic undo mode. *RETENTION* tells Oracle to retain old versions of the *LOB*.

The following example shows our **large_objects** *LOB* table with added parameters to control inline storage and retention of old *LOBs*:

```
CREATE TABLE large_objects
  (pretty_picture BLOB,
   interesting_text CLOB)
STORAGE (INITIAL 256M NEXT 256M)
LOB (pretty_picture, interesting_text)
  STORE AS (TABLESPACE large_object_segment
   STORAGE (INITIAL 512M NEXT 512M)
   NOCACHE LOGGING
   ENABLE STORAGE IN ROW
   RETENTION);
```

The earlier example added parameter values for *STORAGE IN ROW* and *RETENTION*. But since we did not set one for *CHUNK*, that value is set to the Oracle default for the *LOB*.

Oracle nested tables

Oracle allows the declaration of a *NESTED TABLE* in which a table is virtually stored within a column of another table. The *STORE AS* clause enables a proxy name for the table within a table, but the nested table must be created initially as a user-defined datatype. This capability is valuable for sparse arrays of values, but we don't recommend it for day-to-day tasks. For example:

```
CREATE TYPE prop_nested_tbl AS TABLE OF props_nt;  
  
CREATE TABLE proposal_types  
  (proposal_category VARCHAR2(50),  
   proposals          PROPS_NT)  
  
NESTED TABLE props_nt STORE AS props_nt_table;
```

This example creates a table called **proposal_types** along with a nested table called **props_nt**, which is stored as **props_nt_table**.

Oracle compressed tables

Oracle 10g (and Oracle 9i Release 2) allows compression of both keys and entire tables. (Oracle 9i allowed only key compression.) Although compression adds a tiny bit of overhead, it significantly reduces that amount of disk space consumed by a table. This is especially useful for databases pushing the envelope in terms of size. Key compression is handled in the *ORGANIZE INDEX* clause, while table compression is handled in the *ORGANIZE HEAP* clause.

Oracle partitioned and subpartitioned tables

Oracle allows tables to be partitioned and subpartitioned. You can also break out *LOBs* onto their own partition(s). A partitioned table may be broken into distinct parts, possibly placed on separate disk subsystems to improve I/O performance, based on four strategies: range, hash, list, and a composite of the first three. Partitioning syntax is quite elaborate:

```
{ PARTITION BY RANGE (column [...]) (PARTITION [partition_name]  
  VALUES LESS THAN ({MAXVALUE | value} [...])  
  [table_partition_description] ) |  
PARTITION BY HASH (column [...])  
  {(PARTITION [partition_name] [partition_storage_clause] [...]) |  
  PARTITIONS hash_partition_qty [STORE IN (tablespace [...])]  
  [OVERFLOW STORE IN (tablespace [...])]} |  
PARTITION BY LIST (column [...]) (PARTITION [partition_name]  
  VALUES ({MAXVALUE | value} [...])  
  [table_partition_description] ) |  
PARTITION BY RANGE (column [...])  
  {subpartition_by_list | subpartition_by_hash}  
  (PARTITION [partition_name] VALUES LESS THAN  
    ({MAXVALUE | value} [...])  
    [table_partition_description] ) }
```

The following example code shows the **orders** table partitioned by range:

```
CREATE TABLE orders  
  (order_number NUMBER,  
   order_date   DATE,  
   cust_nbr     NUMBER,  
   price        NUMBER,  
   qty          NUMBER,
```



```
    cust_shp_id NUMBER)
PARTITION BY RANGE(order_date)
(PARTITION pre_yr_2000 VALUES LESS THAN
    TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
PARTITION pre_yr_2004 VALUES LESS THAN
    TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
PARTITION post_yr_2004 VALUES LESS THAN
    MAXVALUE) );
```

This example creates three partitions on the **orders** table — one for the orders taken before the year 2000 (**pre_yr_2000**), one for the orders taken before the year 2004 (**pre_yr_2004**), and another for the orders taken after the year 2004 (**post_yr_2004**) — all based on the range of dates that appear in the **order_date** column.

The next example creates the **orders** table based on a hash value in the **customer_shipping_id**:

```
CREATE TABLE orders
(
    order_number NUMBER,
    order_date DATE,
    cust_nbr NUMBER,
    price NUMBER,
    qty NUMBER,
    cust_shp_id NUMBER)
PARTITION BY HASH (cust_shp_id)
(PARTITION shp_id1 TABLESPACE tblspc01,
PARTITION shp_id2 TABLESPACE tblspc02,
PARTITION shp_id3 TABLESPACE tblspc03)
ENABLE ROW MOVEMENT;
```

The big difference in how the records are divided among partitions between the hash partition example and the range partition example is that the range partition code explicitly defines where each record goes, while the hash partition example allows Oracle to decide (by applying a hash algorithm) in which partition to place the record. (Note that we also enabled row movement for the table.)

Examples of list and composite partitioning are shown with the subpartitioning information detailed in the next paragraph. In addition to breaking tables apart into partitions (for easier backup, recovery, or performance reasons), you may further break it apart into subpartitions. The *subpartition_by_list* clause syntax follows:

```
SUBPARTITION BY LIST (column)
[SUBPARTITION TEMPLATE
{ (SUBPARTITION subpartition_name
    [VALUES {DEFAULT | {val | NULL} [,...]}
    [partitioning_storage_clause)] |
    hash_subpartition_qty } ]
```

For example, we'll recreate the **orders** table once again, this time using a range-hash composite partition. In a range-

hash composite partition, the partitions are broken apart by range values, while the subpartitions are broken apart by hashed values. List partitions and subpartitions are broken apart by a short list of specific values. Because you must list out all the values by which the table is partitioned, the partition value is best taken from a small list of values. In this example, we've added a column (**shp_region_id**) that allows four possible regions:

```
CREATE TABLE orders
(
  order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20) )
PARTITION BY RANGE(order_date)
SUBPARTITION BY LIST(shp_region)
SUBPARTITION TEMPLATE(
  (SUBPARTITION shp_region_north
    VALUES ('north','northeast','northwest'),
  SUBPARTITION shp_region_south
    VALUES ('south','southeast','southwest'),
  SUBPARTITION shp_region_central
    VALUES ('midwest'),
  SUBPARTITION shp_region_other
    VALUES ('alaska','hawaii','canada')
(PARTITION pre_yr_2000 VALUES LESS THAN
  TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
PARTITION pre_yr_2004 VALUES LESS THAN
  TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
PARTITION post_yr_2004 VALUES LESS THAN
  MAXVALUE) )
ENABLE ROW MOVEMENT;
```

This code example not only sends the records of the table to one of three partitions based on the **order_date**, it further partitions the records into one of four subpartitions based on the region where the order is being shipped and on the value of the **shp_region** column. By using the *SUBPARTITION TEMPLATE*, you apply the same set of subpartitions to each partition. You can manually override this behavior by specifying subpartitions for each partition.

You may also subpartition using a hashing algorithm. The *subpartition_by_hash* clause follows:

```
SUBPARTITION BY HASH (column [,...])
{SUBPARTITIONS qty [STORE IN (tablespace_name [,...]) ] |
SUBPARTITION TEMPLATE
{ (SUBPARTITION subpartition_name [VALUES {DEFAULT | {val | NULL}
[,...]} [partitioning_storage_clause] ) |
hash_subpartition_qty }
```

The *table_partition_description* clause referenced in the partitioning syntax (in the earlier section [Oracle partitioned and subpartitioned tables](#)) is, in itself, very flexible and supports precise handling of *LOB* and *VARRAY* data:

```
[segment_attr_clause] [[NO] COMPRESS [int]] [OVERFLOW segment_attr_clause]
[partition_level_subpartition_clause]
[ LOB { (lob_item [,...]) STORE AS lob_param_clause |
  (lob_item) STORE AS {lob_segname (log_param_clause) |
  log_segname | (log_param_clause) } } |
VARRAY varray_item {{{[ELEMENT] IS OF [TYPE] (ONLY type_name) |
  [NOT] SUBSTITUTABLE AT ALL LEVELS}} STORE AS LOB { log_segname |
  [log_segname] (log_param_clause) } |
{{{[ELEMENT] IS OF [TYPE] (ONLY type_name) |
  [NOT] SUBSTITUTABLE AT ALL LEVELS}}}
```

The *partition_level_subpartition_clause* syntax follows:

```
{SUBPARTITIONS hash_subpartition_qty [STORE IN (tablespace_name [,...])] |
  SUBPARTITION subpartition_name [VALUES {DEFAULT | {val | NULL} [,...]}
  [partitioning_storage_clause] }
```

The *partition_storage_clause* (see the earlier section [Oracle partitioned and subpartitioned tables](#)), like the table-level *storage_clause* defined earlier, defines how elements of a partition (or subpartition) are stored. The syntax follows:

```
{{TABLESPACE tablespace_name} | [OVERFLOW TABLESPACE tablespace_name] |
VARRAY varray_item STORE AS LOB log_segname |
LOB (lob_item) STORE AS { (TABLESPACE tablespace_name) |
  Log_segname [(TABLESPACE tablespace_name)] }}
```

In this final partitioning example, we'll again recreate the **orders** table using a composite range-hash partition, this time with *LOB* (actually, an *NLOB* column) and storage elements:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NLOB )
PARTITION BY RANGE(order_date)
```

```
SUBPARTITION BY HASH(cust_shp_id)
(PARTITION pre_yr_2000 VALUES LESS THAN
  TO_DATE('01-JAN-2000', 'DD-MON-YYYY') TABLESPACE tblspc01
  LOB (order_desc) STORE AS (TABLESPACE tblspc_a01
    STORAGE (INITIAL 10M NEXT 20M) )
  SUBPARTITIONS subpartn_a,
PARTITION pre_yr_2004 VALUES LESS THAN
  TO_DATE('01-JAN-2004', 'DD-MON-YYYY') TABLESPACE tblspc02
  LOB (order_desc) STORE AS (TABLESPACE tblspc_a02
    STORAGE (INITIAL 25M NEXT 50M) )
  SUBPARTITIONS subpartn_b TABLESPACE tblspc_x07,
PARTITION post_yr_2004 VALUES LESS THAN
  MAXVALUE (SUBPARTITION subpartn_1,
    SUBPARTITION subpartn_2,
    SUBPARTITION subpartn_3
    SUBPARTITION subpartn_4) )
ENABLE ROW MOVEMENT;
```

In this somewhat more complex example, we now define the **orders** table with the added *NLOB* table called **order_desc**. In the **pre_yr_2000** and **pre_yr_2004** partitions, we specify that all of the non-*LOB* data goes to tablespaces **tblspc01** and **tblspc02**, respectively. However, the *NLOB* values of the **order_desc** column will be stored in the **tblspc_a01** and **tblspc_a02**, respectively, with their own unique storage characteristics. Note that the subpartition **subpartn_b** under partition **pre_yr_2004** is also stored in its own tablespace, **tblspc_x07**. Finally, the last partition (**post_yr_2004**) and its subpartitions are stored in the default tablespace for the **orders** table because no partition- or subpartition-level *TABLESPACE* clause overrides the default.

Altering partitioned and subpartition tables

Anything about partitions and subpartitions that is explicitly set by the *CREATE TABLE* statement may be altered after the table is created. Many of the clauses shown later are merely repetitions of clauses that are described in the earlier section about creating partitioned tables; for example, the *SUBPARTITION TEMPLATE* clause or the *MAPPING TABLE* clause. Consequently, these clauses will not be repeated. Altering the partitions and/or subpartitions of an Oracle table is governed by this syntax:

```
ALTER TABLE table_name
[MODIFY DEFAULT ATTRIBUTES [FOR PARTITION partn_name]
  [physical_attributes_clause] [storage_clause] [PCTTHRESHOLD int]
  [{ADD OVERFLOW ... | OVERFLOW ...}] [[NO]COMPRESS]
  [{LOB (lob_name) | VARRAY varray_name} [(lob_parameter_clause)] ]
  [COMPRESS int | NOCOMPRESS]
[SET SUBPARTITION TEMPLATE {hash_subpartn_quantity |
  (SUBPARTITION subpartn_name [partn_list]
  [storage_clause])}]
[MODIFY PARTITION partn_name
  { [table_partition_description] |
```

```
[[REBUILD] UNUSABLE LOCAL INDEXES] |
[ADD [subpartn specification] ] |
[COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause] ] |
[{ADD | DROP} VALUES (partn_value [,...]) ] |
[MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSUED ...} ]
[MODIFY SUBPARTITION subpartn_name {hash_subpartn_attributes |
list_subpartn_attributes} ]
[MOVE {PARTITION | SUBPARTITION} partn_name
[MAPPING TABLE] [table_partition_description] [[NO]PARALLEL]
[update_index_clause] ]
[ADD PARTITION [partn_name] [p table_partition_description]
[[NO]PARALLEL] [update_index_clause] ]
[COALESCE PARTITION [[NO]PARALLEL] [update_index_clause] ]
[DROP {PARTITION | SUBPARTITION} partn_name [[NO]PARALLEL]
[update_index_clause] ]
[RENAME {PARTITION | SUBPARTITION} old_partn_name TO
new_partn_name]
[TRUNCATE {PARTITION | SUBPARTITION} partn_name
[{DROP | REUSE} STORAGE] [[NO]PARALLEL] [update_index_clause] ]
[SPLIT {PARTITION | SUBPARTITION} partn_name {AT | VALUES}
(value [,...])
[INTO (PARTITION [partn_name1]
[table_partition_description] ,
(PARTITION [partn_name2]
[table_partition_description] ) ) ]
[[NO]PARALLEL] [update_index_clause] ]
[MERGE {PARTITION | SUBPARTITION} partn_name1 , partn_name2
[INTO PARTITION [partn_name] [partn_attributes] ] [[NO]PARALLEL]
[update_index_clause] ]
[EXCHANGE {PARTITION | SUBPARTITION} partn_name WITH TABLE table_name
[{INCLUDING | EXCLUDING} INDEXES] [{WITH | WITHOUT} VALIDATION]
[[NO]PARALLEL] [update_index_clause] [EXCEPTIONS INTO table_name] ]
```

where:

MODIFY DEFAULT ATTRIBUTES [FOR PARTITION *partn_name*]

Modifies a wide variety of attributes for the current partition or a specific partition of *partn_name*. Refer to the earlier section [Oracle partitioned and subpartitioned tables](#) for all the details on partition attributes.

SET SUBPARTITION TEMPLATE { *hash_subpartn_quantity* | (SUBPARTITION *subpartn_name* [*partn_list*] [*storage_clause*]) }

Sets a new subpartition template for the table.

MODIFY PARTITION *partn_name*

Changes a wide variety of physical and storage characteristics, including the storage properties of *LOB* and *VARRAY* columns, of a pre-existing partition or subpartition called *partn_name*. Additional syntax may be appended to the *MODIFY PARTITION partn_name* clause:

```
{ [table_partition_description] | [[REBUILD] UNUSABLE LOCAL INDEXES] |  
  [ADD [subpartn_specification] ] |  
  [COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause]  
  { [{UPDATE | INVALIDATE} GLOBAL INDEXES] |  
    UPDATE INDEXES [ (index_name (  
      {index_partn | index_subpartn} ) ) [,...] ] } ] |  
  [{ADD | DROP} VALUES (partn_value [,...] ) ] |  
  [MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...}]
```

where:

table_partition_description

Described in the earlier section [Oracle partitioned and subpartitioned tables](#). This clause may be used on any partitioned table.

[REBUILD] UNUSABLE LOCAL INDEXES

Marks the local index partition as *UNUSABLE*. Adding the optional *REBUILD* keyword tells Oracle to rebuild the unusable local index partition as part of the operation performed by the *MODIFY PARTITION* statement. This clause may not be used with any other subclause of the *MODIFY PARTITION* statement nor may it be used on tables with subpartitions. This clause may be used on any partitioned table.

ADD [*subpartn_specification*]

Adds a hash or list subpartition specification, as described in the earlier section [Oracle partitioned and subpartitioned tables](#), to an existing range partition. This clause may be used to define range-hash or range-list composite partitions only. Oracle populates the new subpartition with rows from other subpartitions using either the hash function or the list values you specify. We recommend the total number of subpartitions be set to a power of 2 for optimal load balancing. You may add range-list subpartitions only if the table does not already have a *DEFAULT* subpartition. When adding range-list subpartitions, the list value clause is required but it cannot duplicate values found in any other subpartition of the current partition. The only storage or physical attribute you may couple with this clause for both range-hash and range-list subpartitions is the *TABLESPACE* clause.

COALESCE SUBPARTITION [[NO]PARALLEL] [*update_index_clause*]

Coalesces the subpartition of a range-hash composite partitioned table. This clause tells Oracle to distribute the contents of the last hash subpartitions to one or more remaining subpartition in the set, and then drop the last hash subpartition. Oracle also drops local index subpartitions corresponding to the subpartition you are coalescing. The *update_index_clause* is described later in this list. Global indexes may be updated or invalidated using the syntax {*UPDATE* | *INVALIDATE*} *GLOBAL INDEXES*. In addition, local indexes, index partitions, or index subpartitions may be updated using the syntax *UPDATE INDEXES* (*index_name* {*index_partn* | *index_subpartn*}).

{ADD | DROP} VALUES (*partn_value* [,...])

Adds a new value (or values) or drops existing values on an existing list-partitioned table, respectively. Local and global indexes are not affected by this clause. Values cannot be added to or dropped from a *DEFAULT* list partition.

MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...}

Defines a mapping table for a partitioned table that is an IOT. The *ALLOCATE EXTENT* and *DEALLOCATE UNUSED* clauses are described earlier in the syntax description list of the "CREATE TABLE" statement. This clause may be used on any type of partitioned table, as long as the table is an index-organized table.

MODIFY SUBPARTITION *subpartn_name* {*hash_subpartn_attributes* | *list_subpartn_attributes*}

Modifies a specific hash or list subpartition according to the subpartition attributes described in the earlier section [Oracle partitioned and subpartitioned tables](#).

[MOVE {PARTITION | SUBPARTITION} *partn_name* [MAPPING TABLE] [*table_partition_description*] [[NO]PARALLEL] [*update_index_clause*]]

Moves a specified partition (or subpartition) of *partn_name* to another partition (or subpartition) described in the *table_partition_description* clause, which is described in the earlier section [Oracle partitioned and subpartitioned tables](#). Moving a partition is I/O intensive, so the optional *PARALLEL* clause may be used to parallelize the operation. When omitted, *NOPARALLEL* is the default. In addition, you may optionally update or invalidate the local and global index as described in the *update_index_clause* shown below in this list.

ADD PARTITION [*partn_name*] [*table_partition_description*] [[NO]PARALLEL]

Adds a new partition (or subpartition) of *partn_name* to the table. The *ADD PARTITION* clause supports all aspects of creating a new partition or subpartition, via the *table_partition_description* clause, which is described in the earlier section [Oracle partitioned and subpartitioned tables](#). Adding a partition may be I/O intensive, so the optional *PARALLEL* clause may be used to parallelize the operation. When omitted, *NOPARALLEL* is the default. In addition, you may optionally update or invalidate local and global indexes on the table using the *update_index_clause* shown earlier in this list.

update_index_clause

Controls the status assigned to indexes once the partitions and/or subpartitions of a table are altered. By default, Oracle invalidates the entire index(es) of a table, not just those portions of the index on the partition and/or subpartition being altered. You may update or invalidate global indexes on the table *OR* update one or more specific index(es) on the table, respectively, using this syntax:

{ UPDATE | INVALIDATE } GLOBAL INDEXES]

UPDATE INDEXES ((*index_name* ({*index_partn*|*index_subpartn*})) [,...] }

COALESCE PARTITION [[NO]PARALLEL] [*update_index_clause*]

Takes the contents of the last partition of a set of hash partitions and reshapes the contents to one or more of the other partitions in the set. The last partition is then dropped. Obviously, this is only for use with hash partitions. The *update_index_clause*, shown earlier in this list, may be applied to update or invalidate the local and/or global indexes of the table being coalesced.

DROP {PARTITION | SUBPARTITION} *partn_name* [[NO]PARALLEL] [*update_index_clause*]

Drops an existing range or list partition or subpartition of *partn_name* from the table. The data within the partition is also dropped. If you want to keep the data, use the *MERGE PARTITION* clause. If you want to get rid of a hash partition or subpartition, use the *COALESCE PARTITION* clause instead. Tables with only a single partition are not affected by the *ALTER TABLE...DROP PARTITION* statement; instead, use the *DROP TABLE* statement.

RENAME {PARTITION | SUBPARTITION} *old_partn_name* TO *new_partn_name*

Renames an existing partition or subpartition of *old_partn_name* to a new name of *new_partn_name*.

TRUNCATE {PARTITION | SUBPARTITION} *partn_name* [{DROP | REUSE} STORAGE] [[NO]PARALLEL]

[*update_index_clause*]

Removes all of the rows of a partition or subpartition of *partn_name*. If you truncate a composition partition, all the rows of the subpartition(s) are also dropped. On IOTs, mapping table partitions and overflow partitions are also truncated. *LOB* data and index segments, if the table has any *LOB* columns, are also truncated. Finally, disable any existing referential integrity constraints on the data, or else delete the rows from the table first, then truncate the partition or subpartition. The optional *DROP STORAGE* and *REUSE STORAGE* subclauses define whether the space freed by the truncated data is made available for other objects in the tablespace or still allocated to the original partition or subpartition, respectively.

```
SPLIT {PARTITION | SUBPARTITION} partn_name {AT | VALUES} (value [...]) [INTO (PARTITION [partn_name1] [table_partition_description], (PARTITION [partn_name2] [table_partition_description]))] [ [NO]PARALLEL] [update_index_clause]
```

Creates from the current partition (or subpartition) of *partn_name* two new partitions (or subpartitions) called *partn_name1* and *partn_name2*. These new partitions may have their own complete specification, as defined by the *table_partition_description* clause. When omitted, the new partitions inherit all physical characteristics of the current partition. When splitting a *DEFAULT* partition, all of the split values go to *partn_name1* while all of the default values go to *partn_name2*. For IOTs, Oracle splits any mapping table partition in a manner corresponding to the split. Oracle also splits *LOB* and *OVERFLOW* segments, but you may also specify your own *LOB* and *OVERFLOW* storage characteristics, described in the earlier section [Oracle partitioned and subpartitioned tables](#). Additional syntax for this clause is described below:

```
{AT | VALUES} (value [...])
```

Splits range partitions (using *AT*) or list partitions (using *VALUES*) according to the *value* or *values* you specify. The *AT* (*value*) clause defines the new noninclusive upper range for the first of the two new partitions. The new value should be less than the partition bound of the current partition, but greater than the partition bound of the next lowest partition (if one exists). The *VALUES* (*value1, value2, [...]*) clause defines the values going into the first of the two new list partitions. The first new list partition is built from *value1* and the second is built from the remaining partition values in the current partition of *partn_name*. The value list must include values that already exist in the current partition, but it cannot contain all of the values of the current partition.

INTO

Defines the two new partitions that result from the split. At a minimum, the two *PARTITION* keywords, in parentheses, are required. Any characteristics not explicitly declared for the new partitions are inherited from the current partition of *partn_name*, including any subpartitioning. There are a few restrictions to note. When subpartitioning range-hash composition partitioned tables, only the *TABLESPACE* value is allowed for the subpartitions. Subpartitioning is not allowed at all when splitting range-list composite partitioned tables. Any indexes on heap-organized tables are invalidated by default when the table is split. You must use the *update_index_clause* to update their status.

```
[MERGE {PARTITION | SUBPARTITION} partn_name1, partn_name2 [INTO PARTITION [partn_name] [partn_attributes]] [ [NO]PARALLEL] [update_index_clause ]
```

Merges the contents of two or more partitions or subpartitions of a table into a single new partition. Oracle then drops the two original partitions. Merged range partitions must be adjacent and are then bound by the higher boundary of the original two partitions when merged. Merged list partitions need not be adjacent and result in a single new partition with a union of the two sets of partition values. If one of the list partitions was the *DEFAULT* partition, then the new partition will be the default. Merged range-list composite partitions are allowed but may not have a new subpartition template. Oracle creates a subpartition template from the existing one(s) or, if none exist, creates a new *DEFAULT* subpartition. Physical attributes not defined explicitly are inherited from the table-level settings. By default, Oracle makes all local index partitions and global indexes *UNUSABLE* unless you override this behavior using the *update_index_clause*. (The exception to this rule is with IOTs, which, being index-based, will remain *USABLE* throughout the merge operation.) Merge operations are not allowed on hash-partitioned tables, so use the *COALESCE PARTITION* clause instead.

```
EXCHANGE {PARTITION | SUBPARTITION} partn_name WITH TABLE table_name [{INCLUDING | EXCLUDING} INDEXES] [{WITH | WITHOUT} VALIDATION] [ [NO]PARALLEL] [update_index_clause] [EXCEPTIONS INTO table_name]
```

Exchanges the data and index segments of a nonpartitioned table with a partitioned table, or of a partitioned table of one type to another partitioned type. The structure of the tables in the exchange must be identical, including the same primary key. All segment attributes (like tablespaces, logging, and statistics) of the current partitioned table, called *partn_name*, and the table it is being exchanged with, called *table_name*, are exchanged. Tables containing *LOB* columns will also exchange *LOB* data and index segments. Additional syntax details, which are not previously defined elsewhere in this list, follow:

WITH TABLE *table_name*

Defines the table that will exchange with the current partition or subpartition.

{*INCLUDING* | *EXCLUDING*} INDEXES

Exchanges local index partitions or subpartitions with the table index (on nonpartitioned tables) or the local index (on hash-partitioned tables), using the *INCLUDING INDEXES* clause. Alternately, marks all index partitions and subpartitions as well as regular indexes and partitioned indexes of the exchanged table to be marked with *UNUSABLE* status, using the *EXCLUDING INDEXES* clause.

{*WITH* | *WITHOUT*} VALIDATION

Returns errors when any rows in the current table fail to map into a partition or subpartition of the exchanged table, using the *WITH VALIDATION* clause. Otherwise, the *WITHOUT VALIDATION* clause may be included to skip checking of row mapping between the tables.

EXCEPTIONS INTO *table_name*

Places the *ROWIDs* of all rows violating a UNIQUE constraint (in DISABLE VALIDATE state) on the partitioned table. When omitted, Oracle assumes a table in the current schema called *EXCEPTIONS*. The *EXCEPTIONS* table is defined in the *utlexcpt.sql* and *utlexpt1.sql* scripts that ship with Oracle. Refer to the Oracle documentation if you need these scripts.

There are a couple of notes to remember about altering a partitioned table. First, altering a partition on a table that serves as the source for one or more materialized views requires that the materialized views be refreshed. Second, bitmap join indexes defined on the partitioned table being altered will be marked *UNUSABLE*. Third, several restrictions apply if the partitions (or subpartitions) are ever spread across tablespaces that use different block sizes. Refer to the Oracle documentation when attempting these sorts of alterations to a partitioned table.

In the next few examples, assume we are using the **orders** table partitioned as shown below:

CREATE TABLE orders

```
(order_number NUMBER,  
order_date DATE,  
cust_nbr NUMBER,  
price NUMBER,  
qty NUMBER,  
cust_shp_id NUMBER)
```

PARTITION BY RANGE(order_date)

```
(PARTITION pre_yr_2000 VALUES LESS THAN  
TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),  
PARTITION pre_yr_2004 VALUES LESS THAN  
TO_DATE('01-JAN-2004', 'DD-MON-YYYY'  
PARTITION post_yr_2004 VALUES LESS THAN  
MAXVALUE) );
```

The following statement will mark all of the local index partitions as *UNUSABLE* in the **orders** table for the **post_yr_2004** partition:

```
ALTER TABLE orders MODIFY PARTITION post_yr_2004  
UNUSABLE LOCAL INDEXES;
```

However, we've decided to now split the **orders** table partition **post_yr_2004** into two all new partitions, **pre_yr_2008** and **post_yr_2008**. Values that are now less than *MAXVALUE* will be stored in the **post_yr_2008** partition, while values less than '01-JAN-2008' will be stored in **pre_yr_2008**:

```
ALTER TABLE orders SPLIT PARTITION post_yr_2004
  AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
  INTO (PARTITION pre_yr_2008, PARTITION post_yr_2008);
```

Assuming that the **orders** table contained a **LOB** or a **VARRAY** column, we could further refine the alteration by including additional details for handling these columns, while also updating the global indexes as the operation completes:

```
ALTER TABLE orders SPLIT PARTITION post_yr_2004
  AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
  INTO
  (PARTITION pre_yr_2008
   LOB (order_desc) STORE AS (TABLESPACE order_tbspc_a1),
  PARTITION post_yr_2008)
  LOB (order_desc) STORE AS (TABLESPACE order_tbspc_a1) )
  UPDATE GLOBAL INDEXES;
```

Now, assuming the **orders** table has been grown at the upper end, lets merge together the partitions at the lower end:

```
ALTER TABLE orders
  MERGE PARTITIONS pre_yr_2000, pre_yr_2004
  INTO PARTITION yrs_2004_and_earlier;
```

After a few years more, we might want to even get rid of the oldest partition, or if not get rid of it, at least give it a better name:

```
ALTER TABLE orders DROP PARTITION yrs_2004_and_earlier;
ALTER TABLE orders RENAME PARTITION yrs_2004_and_earlier TO pre_yr_2004;
```

Finally, let's truncate a table partition, delete all of its data, and return the empty space for use by other objects in the tablespace:

```
ALTER TABLE orders
  TRUNCATE PARTITION pre_yr_2004
  DROP STORAGE;
```

As the examples illustrate, anything relating to partitioning and subpartitioning that can be created with the Oracle *CREATE TABLE* statement can later be changed, augmented, or cut down using the Oracle *ALTER TABLE* statement.

Organized tables: heaps, IOTs, and external tables

Oracle 10g allows powerful means of controlling the physical storage behavior of tables.

The most useful aspect of the *ORGANIZATION HEAP* clause is that you can now compress an entire table within Oracle. This is extremely useful for reducing disk storage costs in database environments with multi-terabyte tables. The following example creates the **orders** table in a compressed and logged heap along with a primary key constraint and storage details:

```
CREATE TABLE orders
(
  order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB,
  CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
ORGANIZATION HEAP
  COMPRESS LOGGING
PCTTHRESHOLD 2
STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
OVERFLOW STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
ENABLE ROW MOVEMENT;
```

To define the same table using an index-organized table based on the **order_date** column, we would use this syntax:

```
CREATE TABLE orders
(
  order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB,
  CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
ORGANIZATION HEAP
  INCLUDING order_date
PCTTHRESHOLD 2
STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
OVERFLOW STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
ENABLE ROW MOVEMENT;
```

Finally, we will create an external table that stores our customer shipping information called **cust_shipping_external**. The code shown in bold is the *opaque_format_spec*:

```
CREATE TABLE cust_shipping_external
  (external_cust_nbr NUMBER(6),
  cust_shp_id      NUMBER,
  shipping_company VARCHAR2(25) )
ORGANIZATION EXTERNAL
  (TYPE oracle_loader
  DEFAULT DIRECTORY dataloader
  ACCESS PARAMETERS
    (RECORDS DELIMITED BY newline
    BADFILE 'upload_shipping.bad'
    DISCARDFILE 'upload_shipping.dis'
    LOGFILE 'upload_shipping.log'
    SKIP 20
    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY ""
    (client_id INTEGER EXTERNAL(6),
    shp_id CHAR(20),
    shipper CHAR(25) ) )
  LOCATION ('upload_shipping.ctl') )
REJECT LIMIT UNLIMITED;
```

Examining this example, we see that the external table type is *ORACLE_LOADER* and the default directory is *DATALOADER*. The example illustrates the fact that you define the metadata of the table within Oracle and then describe how that metadata references a data source outside of the Oracle database server itself.

Oracle XMLType and object-type tables

When an Oracle XMLType table is created, Oracle automatically stores the data in a *CLOB* column, unless you create an XML schema-based table. (For details on Oracle's XML support, see Oracle's XMLDB Developer's Guide.) The following code example first creates an XMLType table, **distributors**, with the implicit *CLOB* data storage, and followed by a second code example, **suppliers**, with a more sophisticated XML-schema definition:

```
CREATE TABLE distributors OF XMLTYPE;
CREATE TABLE suppliers OF XMLTYPE
XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
ELEMENT "vendors";
```

A key advantage of a table based on an XML schema is that you can create B-Tree indexes on them. In the following example, we create an index on **suppliercity**:

```
CREATE INDEX supplier-city-index
ON suppliers
(S:"XMLDATA"."ADDRESS"."CITY");
```

You may similarly create tables using a mix of standard and *XMLTYPE* columns. In this case, the *XMLTYPE* column may store its data as a *CLOB*, or it may store its data in an object-relational column of a structure determined by your specification. For example, we'll recreate the **distributors** table (this time with some added storage specifications) and the **suppliers** table with columns of both standard and *XMLTYPE*:

```
CREATE TABLE distributors
  (distributor_id NUMBER,
   distributor_spec XMLTYPE)
XMLTYPE distributor_spec
STORE AS CLOB
  (TABLESPACE tblspc_dist
   STORAGE (INITIAL 10M NEXT 5M)
   CHUNK 4000
   NOCACHE
   LOGGING);

CREATE TABLE suppliers
  (supplier_id NUMBER,
   supplier_spec XMLTYPE)
XMLTYPE supplier_spec STORE AS OBJECT RELATIONAL
  XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
  ELEMENT "vendors"
OBJECT IDENTIFIER IS SYSTEM GENERATED
OIDINDEX vendor_ndx TABLESPACE tblspc_xml_vendors;
```

When creating XML and object tables, you may refer to *inline_ref_constraint* and *table_ref_constraint* clauses. The syntax for an *inline_ref_constraint* clause is:

```
{SCOPE IS scope_table |
  WITH ROWID |
  [CONSTRAINT constraint_name] REFERENCES object [ (column_name) ]
  [ON DELETE {CASCADE | SET NULL} ]
  [constraint_state] }
```

The only difference between an inline reference constraint and a table reference constraint is that inline reference constraints operate at the column level and table reference constraints operate at the table level, respectively. (This is essentially the same behavior and coding rule as standard relational constraints like *PRIMARY KEY* or *FOREIGN KEY*.) The syntax for a *table_ref_constraint* follows:

```
{SCOPE FOR (ref_col | ref_attr) IS scope_table |
  REF (ref_col | ref_attr) WITH ROWID |
  [CONSTRAINT constraint_name] FOREIGN KEY (ref_col | ref_attr)
  REFERENCES object [ (column_name) ]
  [ON DELETE {CASCADE | SET NULL} ]
  [constraint_state] }
```

The *constraint_state* clause contains a number of options that have already been defined earlier in the Oracle *CREATE TABLE* statement. However, these options are only applied to the condition of the scope reference:

[NOT] DEFERRABLE

INITIALLY {IMMEDIATE | DEFERRED}

{ENABLE | DISABLE}

{VALIDATE | NOVALIDATE}

{RELY | NORELY}

EXCEPTIONS INTO *table_name*

USING INDEX {*index_name* | (*create_index_statement*) | *index_attributes*}

Object-type tables are useful for creating tables containing user-defined types. For example, the following code creates the **building_type** type:

```
CREATE TYPE OR REPLACE building_type AS OBJECT
  (building_name VARCHAR2(100),
  building_address VARCHAR2(200) );
```

We can then create a table called **offices_object_table** that contains the object and defines some of its characteristics, such as OID information. In addition, we'll create two more tables, based upon **building_type**, that reference the object type as an *inline_ref_constraint* and a *table_ref_constraint*, respectively:

```
CREATE TABLE offices_object_table
  OF building_type (building_name PRIMARY KEY)
OBJECT IDENTIFIER IS PRIMARY KEY;
CREATE TABLE leased_offices
  (office_nbr NUMBER,
  rent DEC(9,3),
  office_ref REF building_type
  SCOPE IS offices_object_table);
CREATE TABLE owned_offices
  (office_nbr NUMBER,
  payment DEC(9,3),
  office_ref REF building_type
  CONSTRAINT offc_in_bld REFERENCES offices_object_table);
```

In these examples, the *SCOPE IS* clause defines the *inline_ref_constraint*, while the *CONSTRAINT* clause defines the *table_ref_constraint*.

Oracle ALTER TABLE

When using the Oracle command *ALTER TABLE*, you are able to *ADD*, *DROP*, or *MODIFY* every aspect of each element of the table. For example, the syntax diagram shows that the method for adding or modifying an existing column includes its attributes, but we need to explicitly state that the *attributes* include any Oracle-specific extensions. So while the ANSI standard only lets you modify attributes such as *DEFAULT* or *NOT NULL*, as well as column-level constraints

assigned to the column, Oracle also allows you to alter any special characteristics like *LOB*, *VARRAY*, *NESTED TABLE*, index-organized table, *CLUSTER*, or *PARTITION* settings that might exist.

For example, the following code adds a new column to a table in Oracle, and also adds a new, unique constraint to that table:

```
ALTER TABLE titles
ADD subtitle VARCHAR2(32) NULL
CONSTRAINT unq_subtitle UNIQUE;
```

When a foreign key constraint is added to a table, the DBMS verifies that all existing data in the table meets that constraint. If they do not, the *ALTER TABLE* fails.



Any queries that use **SELECT *** return the new columns, even if this was not planned. Precompiled objects, such as stored procedures, can return any new columns if they use the **%ROWTYPE** attribute. Otherwise, a precompiled object may not return any new columns.

Oracle also allows multiple actions, such as *ADD* or *MODIFY*, to multiple columns to be performed by enclosing the actions within parentheses. For example, the following command adds several columns to a table with a single statement:

```
ALTER TABLE titles
ADD (subtitles VARCHAR2(32) NULL,
     year_of_copyright INT,
     date_of_origin DATE);
```

PostgreSQL

PostgreSQL supports the ANSI standards for *CREATE* and *ALTER TABLE* with a couple of extensions that enable you to quickly build a new table from existing table definitions. Following is the syntax for *CREATE TABLE*:

```
CREATE [LOCAL][TEMPORARY] TABLE table_name
(column_name datatype attributes [...])
| CONSTRAINT constraint_name [{NULL | NOT NULL}]
  {[UNIQUE] | [PRIMARY KEY (column_name [...])] | [CHECK (expression)]
  | REFERENCES reference_table (reference_column [...])
  [MATCH {FULL | PARTIAL | default}]
  [ON {UPDATE | DELETE}
  {CASCADE | NO ACTION | RESTRICT | SET NULL | SET DEFAULT value}]
  [(NOT) DEFERRABLE] [INITIALLY {DEFERRED | IMMEDIATE}] } [...]
| [table_constraint][,...]
[WITH[OUT] OIDE]
[INHERITS (inherited_table [...])]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[AS select_statement]
```

The PostgreSQL syntax for *ALTER TABLE* is:

```
ALTER TABLE [ONLY] table_name [*]  
  
[ADD [COLUMN] column_name datatype attributes [...] ]  
  
| [ALTER [COLUMN] column_name  
    {SET DEFAULT value | DROP DEFAULT | SET STATISTICS int}]  
  
| [RENAME [COLUMN] column_name TO new_column_name]  
  
| [RENAME TO new_table_name]  
  
| [ADD table_constraint]  
  
| [DROP CONSTRAINT constraint_name RESTRICT ]  
  
| [OWNER TO new_owner]
```

The parameters are as follows:

REFERENCES...MATCH...ON{UPDATE | DELETE} ...

Checks a value inserted into the column against the values of a column in another table. It also can be used as part of a *FOREIGN KEY* declaration. The *MATCH* options are *FULL*, *PARTIAL*, and the default where *MATCH* has no other keyword. Full match forces all columns of a multicolumn foreign key either to be NULL or to contain a valid value. The default allows mixed nulls and values. Partial matching is a valid syntax, but is not supported. The *REFERENCES* clause also allows several different behaviors to be declared for *ON DELETE* and/or *ON UPDATE* referential integrity as follows:

NO ACTION

Produces an error when the foreign key is violated (the default).

RESTRICT

A synonym for *NO ACTION*.

CASCADE

Sets the value of the referencing column to the value of the referenced column.

SET NULL

Sets the referencing column value to NULL.

SET DEFAULT *value*

Sets the referencing column to its declared default value or NULL, if no default value exists.

NOT DEFERRABLE [INITIALLY {DEFERRED | IMMEDIATE}]

The *DEFERRABLE* option of the *REFERENCES* clause tells PostgreSQL to defer evaluation of all constraints to the end of a transaction. *NOT DEFERRABLE* is the default behavior for the *REFERENCES* clause. Similar to the *DEFERRABLE* clause is the *INITIALLY* clause. Specifying *INITIALLY DEFERRED* checks constraints at the end of a transaction; *INITIALLY IMMEDIATE* checks constraints after each statement (the default).

FOREIGN KEY

Can be declared only as a table-level constraint and not as a column-level constraint. All options for the *REFERENCES* clause are supported as part of the *FOREIGN KEYS* clause. The syntax follows:

[FOREIGN KEY (*column_name* [...]) REFERENCES...]

WITH[OUT] OIDS

Specifies that rows of the new table should have automatically assigned object IDs (*WITH OIDS*) or not (*WITHOUT OIDS*). The default is *WITH OIDS*. *OIDS* are assigned based on a 32-bit counter. Once the counter is surpassed, the *OID* numbers cycle over and uniqueness can no longer be guaranteed. We recommend that you add a unique index to any *OID* columns you specify to avoid an error with duplicate *OIDS* on very large tables. A table with an inherited structure receives the *OID* setting of its parent table.

INHERITS *inherited_table*

Specifies a table or tables from which the table you are creating inherits all columns. The newly created table also inherits functions attached to tables higher in the hierarchy. If any inherited column appears more than once, the statement fails.

ON COMMIT {DELETE | PRESERVE} ROWS

Used only with temporary tables. This clause controls the behavior of the temporary table after records are committed to the table. *ON COMMIT DELETE ROWS* clears the temporary table of all rows after each commit. This is the default if the *ON COMMIT* clause is omitted. *ON COMMIT PRESERVE ROWS* saves the rows in the temporary table after the transaction has committed.

AS *select_statement*

Enables you to create and populate a table with data from a valid *SELECT* statement. The column names and datatypes do not need to be defined, since they are inherited from the query. The *CREATE TABLE...AS* statement has similar functionality to *SELECT...INTO*, but is more readable.

ONLY

Specifies that only the named table is affected by the *ALTER TABLE* statement and not any parent or subtables in a table hierarchy.

OWNER TO *new_owner*

Changes the owner of the table to the user identified by *new_owner*.

A PostgreSQL table cannot have more than 1,600 columns. However, you should limit the number of columns to well below 1,600 for performance reasons. For example:

CREATE TABLE distributors

```
(name VARCHAR(40) DEFAULT 'Thomas Nash Distributors',  
dist_id INTEGER DEFAULT NEXTVAL('dist_serial'),  
modtime TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```



Unique to PostgreSQL is the ability to create column-level constraints with multiple columns. Since PostgreSQL also supports standard table-level constraints, the ANSI-standard approach is still the recommended approach.

PostgreSQL's implementation of *ALTER TABLE* allows the addition of extra columns using the *ADD* keyword. Existing columns may have new default values assigned to them using *ALTER COLUMN...SET DEFAULT*, while *ALTER COLUMN...DROP DEFAULT* allows the complete erasure of a column-based default. In addition, new defaults may be

added to columns using the *ALTER* clause, but only newly inserted rows will be affected by such new default values. *RENAME* allows new names for existing columns and tables.

SQL Server

SQL Server offers a plethora of options when defining or altering a table, its columns, and its table-level constraints. Its *CREATE TABLE* syntax is:

```
CREATE TABLE table_name
({column_name datatype [ [DEFAULT default_value]
| {IDENTITY [(seed,increment) [NOT FOR REPLICATION]]]
| ROWGUIDCOL][NULL | NOT NULL]
| [{PRIMARY KEY | UNIQUE}
| [CLUSTERED | NONCLUSTERED]
| [WITH FILLFACTOR = inf] [ON {filegroup | DEFAULT}] ]
| [[FOREIGN KEY]
REFERENCES reference_table(reference_column[,...])
| ON {DELETE | UPDATE} {CASCADE | NO ACTION}]
| [NOT FOR REPLICATION]
| [CHECK [NOT FOR REPLICATION] (expression)
| [COLLATE collation_name]
| column_name AS computed_column_expression }
[,...]
| [table_constraint] [,...] )
[ON {filegroup | DEFAULT} ]
[TEXTIMAGE_ON {filegroup | DEFAULT} ]
```

The SQL Server version of *ALTER TABLE* is:

```
ALTER TABLE table_name
[ALTER COLUMN column_name new_datatype attributes {ADD | DROP} ROWGUIDCOL]
| [ADD [COLUMN] column_name datatype attributes [,...]
| [WITH CHECK | WITH NOCHECK] ADD table_constraint [,...]
| [DROP { [ CONSTRAINT ] constraint_name | COLUMN column_name }] [,...]
| [{ CHECK | NOCHECK } CONSTRAINT { ALL | constraint_name [,...] }]
| [{ ENABLE | DISABLE } TRIGGER { ALL | trigger_name [,...] }]
```

The parameters are as follows:

IDENTITY [(*seed, increment*)]

Creates and populates the column with a monotonically increasing number when applied to an integer column. The *IDENTITY* starts counting at the value of *seed* and increases by the value of *increment*. When either are omitted, the default is 1.

NOT FOR REPLICATION

Specifies that the values of an *IDENTITY* or *FOREIGN KEY* are not replicated to subscribing servers. This helps in situations in which different servers require the same table structures, but not the exact same data.

ROWGUIDCOL

Identifies a column as a global unique identifier, which ensures no two values are ever repeated across any number of servers. Only one such column may be identified per table. It does not, however, create the unique values itself. They must be inserted using the *NEWID* function.

DEFAULT

Applies to any column except those with a *TIMESTAMP* datatype or an *IDENTITY* property. The *DEFAULT* must be a constant value such as a character string or a number, and a system function such as *GETDATE()* or *NULL*.

PRIMARY KEY

Defines a primary key constraint for the table. The primary key declaration differs from the ANSI standard by allowing you to assign the clustered or nonclustered attributes on the primary key index, as well as a starting fillfactor. (Refer to the [CREATE/ALTER INDEX Statement](#) for more information.) The attributes of a primary key include:

CLUSTERED

Declares that the column or columns of the primary key set the physical sort order of the records in the table. *CLUSTERED* is the default when the clustered or nonclustered keywords are omitted.

NONCLUSTERED

Declares that the primary key index maintains pointers to all of the records of the table.

WITH FILLFACTOR = int

Declares that a percentage of space, *int*, should remain free on each data page when the table is created. SQL Server does not maintain *FILLFACTOR* over time. You should rebuild the index on a regular basis.

ON {filegroup | DEFAULT}

Specifies that the primary key is located either on the pre-existing, named filegroup or is assigned to the *DEFAULT* filegroup.

FOREIGN KEY

Checks values as they are inserted into the table against a column in another table in order to maintain referential integrity. Foreign keys are described in detail in [Chapter 2](#). A foreign key can only reference columns that are defined as a *PRIMARY KEY* or *UNIQUE* index on the referencing table. A referential action may be specified to take place on the *reference_table* when the record is deleted or updated, according to the following:

ON {DELETE | UPDATE}

Specifies that an action needs to happen in the local table when either (or both) an *UPDATE* or *DELETE* occur on the referenced table.

CASCADE

Specifies that any *DELETE* or *UPDATE* also takes place on the referring table to any records dependent on the value of the *FOREIGN KEY*.

NO ACTION

Specifies that no action occur on the referring table when a record on the current table is deleted or updated.

NOT FOR REPLICATION

Specifies that an *IDENTITY* property should not be enforced when the data is replicated from another database. This ensures that data from the published server are not assigned new identity values.

CHECK

Ensures that a value inserted into the specified column of the table is a valid value based on the *CHECK* expression. For example, the following shows a table with two column-level *CHECK* constraints.

CREATE TABLE people

```
(people_id CHAR(4)
    CONSTRAINT pk_dist_id PRIMARY KEY CLUSTERED
    CONSTRAINT ck_dist_id CHECK (dist_id LIKE '
[A-Z][A-Z][A-Z][A-Z]),
people_name VARCHAR(40) NULL,
people_addr1 VARCHAR(40) NULL,
people_addr2 VARCHAR(40) NULL,
city VARCHAR(20) NULL,
state CHAR(2) NULL
    CONSTRAINT def_st DEFAULT ("CA")
    CONSTRAINT chk_st REFERENCES states(state_ID),
zip CHAR(5) NULL
    CONSTRAINT ck_dist_zip
CHECK(zip LIKE '[0-9][0-9][0-9][0-9]'),
phone CHAR(12) NULL,
sales_rep empid NOT NULL DEFAULT USER)
```

GO

The *CHECK* constraint on the **people_id** ensures an all-alphabetic ID, while the one on *zip* ensures an all-numeric value. The *REFERENCES* constraint on *state* performs a look-up on the **states** table. The *REFERENCES* constraint is essentially the same as a *CHECK* constraint, except that it derives its list of acceptable values from the values stored in another column. This example illustrates how column-level constraints are named using the *CONSTRAINT constraint_name* syntax.

COLLATE

Allows programmers to change, on a column-by-column basis, the sort order and character set that is used by the column.

TEXTIMAGE_ON {filegroup | DEFAULT}

Controls the placement of **text**, **ntext**, and **image** columns allowing you to place *LOB* data on the pre-existing *filegroup* of your choice. When omitted, these columns are stored in the default filegroup with all other tables and database objects.

WITH [NO]CHECK

Tells SQL Server whether the data in the table should be validated against any newly added constraints or keys. When constraints are added using *WITH NOCHECK*, the query optimizer ignores them until they are enabled via *ALTER TABLE table_name CHECK CONSTRAINT ALL*. When constraints are added using *WITH CHECK*, the constraints are checked immediately against all data already in the table.

[NO]CHECK CONSTRAINT

Enables an existing constraint with *CHECK CONSTRAINT* or disables one with *NOCHECK CONSTRAINT*.

```
{ENABLE | DISABLE} TRIGGER { ALL | trigger_name [,...] }
```

Enables or disables the specified trigger or triggers, respectively. All triggers on the table may be enabled or disabled by substituting the keyword *ALL* for the table name, as in *ALTER TABLE employee DISABLE TRIGGER ALL*. You may, alternately, disable or enable a single *trigger_name* or more than one trigger by placing each *trigger_name* in a comma-delimited list.

SQL Server allows any column-level constraint to be named by specifying *CONSTRAINT constraint_name...*, and then the text of the constraint. Several constraints may be applied to a single column, as long as they are not mutually exclusive (for example, *PRIMARY KEY* and *NULL*).

SQL Server also allows a local temporary table to be created, but not using ANSI syntax. A local temporary table, which is stored in the **tempdb** database, requires a prefix of a single pound sign (**#**) to the name of the table. The local temporary table is usable by the person or process that created it, and is deleted when the person logs out or the process terminates. A global temporary table, which is usable to all people and processes that are currently logged in, can be established by prefixing two pound signs (**##**) to the name of the table. The global temporary table is deleted when its process terminates or its creator logs out.

SQL Server also allows the creation of tables with columns that contain a computed value. Such a column does not actually contain data. Instead, it is a virtual column containing an expression using other columns already in the table. For example, a computed column could have an expression, such as *order_cost AS (price * qty)*. Computed columns also can be constants, functions, variables, noncomputed columns, or any of these combined with each other using operators.

Any of the column-level constraints shown earlier also may be declared at the table level. That is, *PRIMARY KEY* constraints, *FOREIGN KEY* constraints, *CHECK* constraints, and others may be declared after all the columns have been defined in the *CREATE TABLE* statement. This is very useful for constraints that cover more than one column. For example, a column-level *UNIQUE* constraint can be applied only to that column. However, declaring the constraint at the table level allows it to span several columns. Here is an example of both column-level and table-level constraints:

-- Creating a column-level constraint

```
CREATE TABLE favorite_books
```

```
(isbn      CHAR(100) PRIMARY KEY NONCLUSTERED,  
 book_name VARCHAR(40) UNIQUE,  
 category  VARCHAR(40) NULL,  
 subcategory VARCHAR(40) NULL,  
 pub_date  DATETIME NOT NULL,  
 purchase_date DATETIME NOT NULL)
```

```
GO
```

-- Creating a table-level constraint

```
CREATE TABLE favorite_books
```

```
(isbn      CHAR(100) NOT NULL,  
 book_name VARCHAR(40) NOT NULL,  
 category  VARCHAR(40) NULL,  
 subcategory VARCHAR(40) NULL,  
 pub_date  DATETIME NOT NULL,  
 purchase_date DATETIME NOT NULL,  
 CONSTRAINT pk_book_id PRIMARY KEY NONCLUSTERED (isbn)  
 WITH FILLFACTOR=70,
```

```
CONSTRAINT unq_book UNIQUE CLUSTERED (book_name, pub_date)
```

```
GO
```

These two commands provide nearly the same results, except that the table-level *UNIQUE* constraint has two columns, whereas only one column is included in the column-level *UNIQUE* constraint.

The following example adds a new *CHECK* constraint to a table, but does not check to ensure that the existing values in the table pass the constraint:

```
ALTER TABLE favorite_book WITH NOCHECK
```

```
ADD CONSTRAINT extra_check CHECK (ISBN > 1)
```

```
GO
```

In this example, we further add a new column with an assigned *DEFAULT* value that is placed in each existing record in the table:

```
ALTER TABLE favorite_book ADD reprint_nbr INT NULL
```

```
CONSTRAINT add_reprint_nbr DEFAULT 1 WITH VALUES
```

```
GO
```

```
-- Now, disable the constraint
```

```
ALTER TABLE favorite_book NOCHECK CONSTRAINT add_reprint_nbr
```

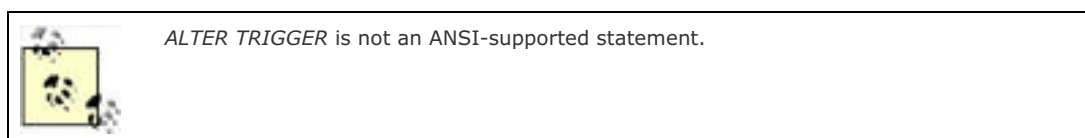
```
GO
```

See Also

```
CREATE SCHEMA  
DROP
```

CREATE/ALTER TRIGGER Statement

A *trigger* is a special kind of stored procedure that fires automatically (hence the term "trigger") when a specific data-modification statement is executed against a table. The trigger is directly associated with the table and is considered a dependent object. For example, you might want all of the **part_numbers** in the **sales** table to be updated when a **part_number** is changed in the **products** table, thus ensuring that part numbers are always in sync.



Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {DELETE | INSERT | UPDATE [OF column [,...]]}
ON table_name
[REFERENCING {OLD {[ROW] | TABLE} [AS] old_name | NEW
{ROW | TABLE} [AS] new_name}] [FOR EACH { ROW | STATEMENT }]
[WHEN (conditions)]
[BEGIN ATOMIC]
code_block
[END]
```

Keywords

CREATE TRIGGER *trigger_name*

Creates a trigger of *trigger_name* and associates it with a specific table.

BEFORE | *AFTER*

Declares that the trigger logic is fired either *BEFORE* or *AFTER* the data manipulation operation that invoked the trigger. *BEFORE* triggers perform their operations before the *INSERT*, *UPDATE*, or *DELETE* operation occurs, allowing you to do dramatic things like circumvent the data manipulation operation altogether. *AFTER* triggers fire after the data manipulation operation has completed, but are useful for after-the-fact operations like recalculating running totals.

DELETE | INSERT | UPDATE [OF *column* [,...]]

Defines the data manipulation operation that causes the trigger to fire: *DELETE* statements, *INSERT* statements, or *UPDATE* statements. You may optionally chose which columns will trigger an update trigger using *UPDATE OF column* [,...]. If an update occurs on any columns not in the column list, the trigger will not fire.

ON *table_name*

Declares the pre-existing table that the trigger is dependent on.

REFERENCING {OLD {[ROW] | TABLE} [AS] *old_name* | NEW {ROW | TABLE} [AS] *new_name*}

Enables aliasing for the old or new *ROW* or *TABLE* acted upon by the trigger. Although the syntax shows the options as exclusive, you may have up to four aliasing references: one for old row, one for old table, one for new row, and one for new table. The alias *OLD* refers to the data contained in the table or row before the data manipulation operation that fired the trigger, while the alias *NEW* refers to the data that would be contained in the table or row after the data manipulation operation that fired the trigger. Note that the syntax indicates that *ROW* is optional, but *TABLE* is not. (That is, *OLD ROW AS* is the same as *OLD AS*, but for *TABLE*, the only valid option is *OLD TABLE AS*.) *INSERT* triggers do not have an *OLD* context, while *DELETE* triggers do not have a *NEW* context. The keyword *AS* is noise and may be omitted. If the *REFERENCING* clause specifies either *OLD ROW* or *NEW ROW*, then the *FOR EACH ROW* clause is required.

FOR EACH { ROW | STATEMENT }

Tells the database to apply the trigger for each row in the table that has changed (*ROW*) or for each SQL

statement issued against the table (*STATEMENT*). For example, a single *UPDATE* statement will update the salaries of 100 employees. If you specify *FOR EACH ROW*, the trigger will execute 100 times. If you specify *FOR EACH STATEMENT*, the trigger will execute only once.

WHEN (*conditions*)

Allows you to define additional criteria for a trigger. For example, you might have a trigger called *DELETE employee*, which will fire whenever an employee is deleted. When a trigger fires, if the search conditions contained in the *WHEN* clause evaluate to TRUE, the trigger action will fire. Otherwise, the trigger will not fire.

BEGIN ATOMIC | *code_block* | END

The ANSI standard requires that the *code_block* should contain only one SQL statement or, for multiple SQL statements, put them in a *BEGIN* and *END* block.

Rules at a Glance

Triggers, by default, fire once at the *statement level*. That is, a single *INSERT* statement might insert 500 rows into a table, but an insert trigger on that table fires only one time. Some vendors allow a trigger to fire for each row of the data-modification operation. So, a statement that inserts 500 rows into a table that has a row-level insert trigger fires 500 times, once for each inserted row.

In addition to being associated with a specific data-modification statement (*INSERT*, *UPDATE*, or *DELETE*) on a given table, triggers are associated with a specific *time* of firing. In general, triggers can fire *BEFORE* the data-modification statement is processed, *AFTER* it is processed, or (when supported by the vendor) *INSTEAD OF* processing the statement. Triggers that fire before or instead of the data-modification statement do not see the changes that the statement renders, while those that fire afterwards can see and act upon the changes that the data-modification statement renders.

Triggers make use of two *pseudotables*. They are pseudotables in the sense that they are not declared with a *CREATE TABLE* statement, but they exist logically on the database. The pseudotables have different names on the different platforms, but we'll call them **Before** and **After** here. They are structured exactly the same as the table a trigger is associated with. However, they contain snapshots of the table's data. The **Before** table contains a snapshot of all the records in the table before the trigger fired, while the **After** table contains a snapshot of all the records in the table after the event occurred that fired the trigger. You can then use comparison operations to compare the data in the table before and after the event fired the trigger to determine exactly what you want to happen.

Here is an Oracle *BEFORE* trigger that uses the *OLD* and *NEW* pseudotables to compare values. (By way of comparison, SQL Server uses the *DELETED* and *INSERTED* pseudotables in the same way, while DB2 requires that you provide a *REFERENCING* clause to give names to the pseudotables.) This trigger creates an audit record before changing an employee's pay record:

```
CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
WHEN (new.emp_salary <> old.emp_salary)
BEGIN
    INSERT INTO employee_audit
    VALUES ('old', :old.emp_id, :old.emp_salary, :old.emp_ssn);
END;
```

You can also take advantage of capabilities within each database platform to make your triggers more powerful and easier to program. For example, Oracle has a special *IF...THEN* clause for use just in triggers. This *IF...THEN* clause takes the form: *IF DELETING THEN*, *IF INSERTING THEN*, or *IF UPDATING THEN*. The following example builds an Oracle *DELETE* and *UPDATE* trigger that uses the *IF DELETING THEN* clauses:

```
CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
BEGIN
    IF DELETING THEN
```



```
INSERT INTO employee_audit
VALUES ('DELETED', :old.emp_id, :old.emp_salary, :old.emp_ssn);

ELSE

INSERT INTO employee_audit
VALUES ('UPDATED', :old.emp_id, :new.emp_salary, :old.emp_ssn);

END IF;

END;
```

This SQL Server example adds a new table called **contractor** to the database. All records in the **employee** table that indicate that the employee is a contractor were moved into the **contractor** table. Now all new employees inserted into the **employee** table will go into the **contractor** table instead through an *INSTEAD OF INSERT* trigger:

```
CREATE TRIGGER if_emp_is_contractor
INSTEAD OF INSERT ON employee
BEGIN
INSERT INTO contractor
SELECT * FROM inserted WHERE status = 'CON'
INSERT INTO employee
SELECT * FROM inserted WHERE status = 'FTE'
END
GO
```

Adding triggers to a table that already has data in it does not cause the triggers to fire. A trigger will fire only for data-modification statements declared in the trigger definition that occur after the trigger is created.

A trigger generally ignores structural changes to tables, such as an added column or an altered datatype on an existing column, after the trigger has been created unless a table modification directly interferes with the operation of the trigger. For example, adding a new column to a table will result in any existing trigger ignoring the new column, except for *UPDATE* triggers. On the other hand, removing a column from a table that is used by a trigger will cause the trigger to fail every time it executes.

Programming Tips and Gotchas

One of the key programming issues associated with triggers is the inappropriate and uncontrolled use of *nested triggers* and *recursive triggers*. A nested trigger is a trigger that invokes a data-manipulation operation that causes other triggers to fire. For example, assume we have three tables, **T1**, **T2**, and **T3**. Table **T1** has a *BEFORE INSERT* trigger that inserts a record into table **T2**. Table **T2** also has a *BEFORE INSERT* trigger that inserts a record into table **T3**. Although this is not necessarily a bad thing if your logic is well considered and fully thought out, it does introduce two problems. First, an insert into table **T1** requires many more I/O operations and transactions than a simple *INSERT* statement against table **T1**. Second, you can get yourself into hot water if, in fact, table **T3** performs an *INSERT* operation against table **T1**. In a case like that, you may have a looping trigger process that can consume all available disk space and even shut down the server.

Recursive triggers are triggers that can fire themselves; for example an *INSERT* trigger that performs an *INSERT* against its own base table. If the procedural logic within the *code_body* is not properly constructed, you can cause a looping trigger error. However, recursive triggers often require setting a special configuration flag before they can be used on the various database platforms.

DB2

DB2 supports a superset of the ANSI SQL standard for the *CREATE TRIGGER* statement. It *does not* support an *ALTER TRIGGER* statement. The *CREATE TRIGGER* syntax is:

```
CREATE TRIGGER trigger_name
```

```
{NO CASCADE BEFORE | AFTER | INSTEAD OF}  
{[DELETE] | [INSERT] | [UPDATE [OF (column [,...]) ]]}  
ON table_name  
[REFERENCING  
  {OLD[_TABLE] [AS] old_name | NEW[_TABLE]  
  [AS] new_name} DEFAULTS NULL]  
[FOR EACH { ROW | STATEMENT }]  
MODE DB2SQL  
[WHEN (conditions)  
code_block
```

where:

NO CASCADE BEFORE

Declares that the trigger action is applied *before* the data-modification operation is applied to the table. This clause also declares that the trigger action will not cause any other triggers to fire.

ON *table_name*

Specifies the pre-existing table or view that the trigger is associated with. Triggers can be declared on regular tables, but not on temporary tables, nicknames, catalog tables, or materialized query tables. When creating triggers on a view, only *INSTEAD OF* triggers are allowed. The view must be a non-typed view that does not use the *WITH CHECK OPTION* clause. The data-modification operation applied to the view should be translated against the base table to determine whether it will fire the trigger or not.

OLD[_TABLE] [AS] *old_name* | NEW[_TABLE] [AS] *new_name*

Defines the alias for the ANSI *before* and *after* pseudotables using *OLD_TABLE* and *NEW_TABLE*, respectively. If you leave off the *_TABLE* keyword, then DB2 assigns the trigger operation as row-level, and allows you to access the row (singular) data prior to triggering the SQL operation. Adding in the *_TABLE* keyword declares that the trigger operation allows you to reference the set (plural) of row data prior to triggering the SQL operation. *OLD* and *OLD_TABLE* may only be used on *DELETE* or *UPDATE* triggers. Conversely, *NEW* and *NEW_TABLE* can only be used on *INSERT* and *UPDATE* triggers. *OLD* and *NEW* may only be specified once each. *OLD_TABLE* and *NEW_TABLE* may only be specified once each and not on *NO CASCADE BEFORE* triggers.

DEFAULTS

Inserts default values where they have been created.

FOR EACH { ROW | STATEMENT }

Specifies that the trigger action applies either at the row level or the statement level. The *FOR EACH STATEMENT* clause is only allowed on *AFTER* triggers.

MODE DB2SQL

Declares that the mode of the trigger is DB2 SQL, the only mode currently supported.

WHEN (*conditions*)

Meets the ANSI SQL standard with the exception that this clause is not allowed on DB2 *INSTEAD OF* triggers.

DB2 does not prevent a new trigger from being created that has the same definition as an older trigger. Thus, it's possible to have two (or more) *AFTER DELETE* triggers on a single table. In a situation like this, DB2 will always fire the

triggers in the order they were created.

Be careful declaring a *BEFORE DELETE* trigger on a table with a self-referencing constraint and a delete rule of *CASCADE*. Such a trigger may produce results that you did not expect!



DB2 allows the *COMMENT ON object_type* statement, so that you can provide extra commentary on any supported DB2 object. For example, you can issue the *COMMENT ON TRIGGER trigger_name* statement to add comments on any trigger you add to the database. You can use this on tables, views, etc.

DB2 does not specifically drop the triggers associated with a table or view when the table or view itself is dropped. Consequently, you should first issue a *DROP TRIGGER* statement followed by a *DROP TABLE* statement to avoid leaving orphaned triggers behind. Conversely, triggers are not reinstated if a table is dropped and then recreated. The triggers should be recreated as well.

BEFORE and *AFTER* triggers are allowed on typed tables at any level of a typed-table hierarchy. Triggers may activate on rows of subtables. If a SQL statement activates multiple triggers, the triggers will be executed in their creation order, even if they are attached to different tables in the typed-table hierarchy.

MySQL

Not supported.

Oracle

Oracle supports the ANSI standard for *CREATE TRIGGER*, with several additions and variations:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{ [{object_event] [database_event] [...] ON {DATABASE | schema.SCHEMA} } |
  { [DELETE] [OR] [INSERT] [OR] [UPDATE [OF column [,...]]] [...] }
  ON {table_name | [NESTED TABLE column_name OF] view_name}
  [REFERENCING {[OLD [AS] old_name] [NEW [AS] new_name]
  [PARENT [AS] parent_name]}
  [FOR EACH ROW] }
[WHEN (conditions)]
code_block
```

The syntax for *ALTER TRIGGER*, which allows you to rename, enable, or disable a trigger without dropping and recreating it, is shown below:

```
ALTER TRIGGER trigger_name
{ {ENABLE | DISABLE} | RENAME TO new_name |
  COMPILE [compiler_directives] [DEBUG] [REUSE SETTINGS]}
```

where:

OR REPLACE

Recreates an existing trigger by assigning a new definition to an existing trigger of *trigger_name*.

object_event

In addition to the standard data-modification events, Oracle allows triggers to fire based on object events. *object_event* operations may be paired with *BEFORE* and *AFTER* keywords. An *object_event* fires the trigger whenever such an event occurs, according to the following keywords:

ALTER

Fires whenever an *ALTER* statement (except *ALTER DATABASE*) is issued.

ANALYZE

Fires whenever Oracle validates the structure of a database object, or collects or deletes statistics on an index.

ASSOCIATE STATISTICS

Fires whenever Oracle associates a statistics type with a database object.

AUDIT

Fires whenever Oracle tracks a SQL statement or operation against a schema object.

COMMENT

Fires whenever an Oracle comment is added in the data dictionary to a database object.

DDL

Fires whenever Oracle encounters any *object_event* in this list.

DISASSOCIATE STATISTICS

Fires whenever Oracle disassociates a statistics type from a database object.

DROP

Fires whenever a *DROP* statement erases a database object from the data dictionary.

GRANT

Fires whenever a user grants privileges or roles to another user or role.

NOAUDIT

Fires whenever the *NOAUDIT* statement causes Oracle to stop tracking SQL statements or operations against schema objects.

RENAME

Fires whenever the *RENAME* statement changes the name of a database object.

REVOKE

Fires whenever a user revokes privileges or roles from another user or role.

TRUNCATE

Fires whenever a *TRUNCATE* statement is issued against a table or cluster.

database_event

In addition to the standard data-modification events, Oracle allows triggers to fire based on database events. *Database_event* operations may be paired with *BEFORE* and *AFTER* keywords. The list of allowable *database_event* keywords is:

LOGON

Fires whenever a client application logs on to the database. Valid for *AFTER* triggers only.

LOGOFF

Fires whenever a client application logs off of the database. Valid for *BEFORE* triggers only.

SERVERERROR

Fires whenever a server error message is logged. Valid for *AFTER* triggers only.

SHUTDOWN

Fires whenever an instance of the database is shut down. Valid for *BEFORE* triggers only with the *ON DATABASE* clause only.

STARTUP

Fires whenever an instance of the database is opened. Valid for *AFTER* triggers with the *ON DATABASE* clause.

SUSPEND

Fires whenever a server error causes a transaction to suspend. Valid for *AFTER* triggers only.

ON {DATABASE | schema.SCHEMA}

Declares that the trigger fires whenever any database user invokes a triggering event with *ON DATABASE*. The trigger then fires for events occurring anywhere in the entire database. Otherwise, *ON schema.SCHEMA* declares that the trigger fires whenever a user connected as *schema* invokes a triggering event. The trigger then fires for events occurring anywhere in the current schema.

ON [NESTED TABLE column_name OF] view_name

Declares that the trigger fires only if the data-manipulation operation applies to the column(s) of the view called *view_name*. The *ON NESTED TABLE* clause is compatible only with *INSTEAD OF* triggers.

REFERENCING PARENT [AS] parent_name

Defines the alias for the current row of the parent table (i.e., supertable). Otherwise, identical to the ANSI standard.

ENABLE

Enables a deactivated trigger when used with *ALTER TRIGGER*. You may alternately use the statement *ALTER TABLE table_name ENABLE ALL TRIGGERS*.

DISABLE

Disables an activated trigger when used with *ALTER TRIGGER*. You may alternately use the statement *ALTER TABLE table_name DISABLE ALL TRIGGERS*.

RENAME TO *new_name*

Renames the trigger to *new_name*, though the state of the trigger remains unchanged when used with *ALTER TRIGGER*.

COMPILE [DEBUG] [REUSE SETTINGS]

Compiles a trigger, whether valid or invalid, and all the objects that the trigger depends on. If any the objects are invalid, then the trigger is invalid. If all of the objects are valid, including the *code_body* of the trigger, then the trigger becomes valid. The subclauses of the *COMPILE* clause are:

DEBUG

Tells the PL/SQL compiler to generate and store the extra information for use by the PL/SQL debugger.

REUSE SETTINGS

Tells Oracle to retain all compiler switch settings, which can save significant time during the compile process.

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive* = '*value*'. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

When referencing values in the *OLD* and *NEW* pseudotables, the values must be prefaced with a colon (:), except in the trigger's *WHEN* clause, where no colons are used. In this example, we'll call a procedure in the *code_body* and use both *:OLD* and *:NEW* values as arguments:

```
CREATE TRIGGER scott.sales_check
BEFORE INSERT OR UPDATE OF ord_id, qty ON scott.sales
FOR EACH ROW
WHEN (new.qty > 10)
CALL check_inventory(:new.ord_id, :new.qty, :old.qty);
```

Multiple trigger types may be combined into a single trigger command *if* they are of the same level (row or statement) and they are on the same table. When triggers are combined in a single statement, the clauses *IF INSERTING THEN*, *IF UPDATING THEN*, and *IF DELETING THEN* may be used in the PL/SQL block to break the code logic into distinct segments. An *ELSE* clause also can be used in this structure.

Following is an example of a *database_event*-style trigger:

```
CREATE TRIGGER track_errors
AFTER SERVERERROR ON DATABASE
BEGIN
IF (IS_SERVERERROR (04030))
THEN INSERT INTO errors ('Memory error');
ELSE (IS_SERVERERROR (01403))
THEN INSERT INTO errors ('Data not found');
END IF;
END;
```

In another example, we can create a trigger that is *SCHEMA*-wide in scope:

```
CREATE OR REPLACE TRIGGER create_trigger
AFTER CREATE ON scott.SCHEMA
BEGIN
    RAISE_APPLICATION_ERROR (num => -20000, msg =>
        'Scott created an object');
END;
```

PostgreSQL

PostgreSQL's implementation of *CREATE TRIGGER* offers a subset of features found in the ANSI standard. On PostgreSQL, a trigger may fire *BEFORE* a data-modification operation is attempted on a record and before any constraints are fired or it may fire *AFTER* a data-manipulation operation fires (and after constraints have been checked), making all operations involved in the transaction visible to the trigger. Finally, the trigger may fire *INSTEAD OF* the data-modification operation and completely replace the *INSERT*, *UPDATE*, or *DELETE* statement with some other behavior. The *CREATE TRIGGER* syntax is:

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER }
{ {[DELETE] [OR | ,] [INSERT] [OR | ,] [UPDATE]} [OR ...] }
ON table_name
FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE function_name (parameters)
```

The parameters are:

OR

Declares an additional trigger action. The *OR* keyword is a synonym for the comma delimiter.

FOR EACH ROW

Explicitly declares that the trigger acts on the row level. This is also the default. Although PostgreSQL does not reject the *STATEMENT* keyword, it does not enforce statement-level triggers.

EXECUTE PROCEDURE *function_name* (*parameters*)

Executes a previously defined function (created using *CREATE FUNCTION*) rather than a block of procedural code. (PostgreSQL does not have its own procedural language.)

The following is an example of a PostgreSQL *BEFORE* trigger that checks at a row level to ensure that the specified distributor code exists in the **distributors** table before inserting or updating a row in the **sales** table:

```
CREATE TRIGGER if_dist_exists
BEFORE INSERT OR UPDATE ON sales
FOR EACH ROW
EXECUTE PROCEDURE check_primary_key
    ('dist_id', 'distributors', 'dist_id');
```

BEFORE and *AFTER* triggers are supported by the ANSI standard. *INSTEAD OF* triggers on PostgreSQL completely skip the data-modification operation that triggered them in favor of code that you substitute for the data-modification operation.

SQL Server

SQL Server supports the basics of the ANSI standard with the addition of the *INSTEAD OF* trigger type and a column change check. It does not support the *REFERENCING* or *WHEN* clauses. Its syntax is:

```
{CREATE | ALTER} TRIGGER trigger_name ON table_name
[WITH ENCRYPTION]
{FOR | AFTER | INSTEAD OF} {[DELETE] [,] [INSERT] [,] [UPDATE]}
[WITH APPEND]
[NOT FOR REPLICATION]
AS
[IF UPDATE(column) [{AND | OR} UPDATE(column)][... ]
code_block
```

where:

```
{CREATE | ALTER} TRIGGER trigger_name
```

Creates a new trigger of *trigger_name* or alters an existing trigger of *trigger_name* by adding or changing trigger properties or the trigger *code_block*. When altering an existing trigger, the permissions and dependencies of the existing trigger are maintained.

```
ON table_name
```

Declares the table or view that the trigger is dependent on. Views may have *INSTEAD OF* triggers defined on them, as long as they are updateable and do not have the *WITH CHECK* clause on them.

```
WITH ENCRYPTION
```

Encrypts the text of the *CREATE TRIGGER* statement as shown in the **syscomments** table. This option is useful to protect important intellectual property. *WITH ENCRYPTION* prevents the trigger from use in a SQL Server replication scheme.

```
FOR | AFTER | INSTEAD OF
```

Tells SQL Server when the trigger should fire. (*FOR* and *AFTER* are synonyms.) *AFTER* specifies that the trigger fire only after the triggering data-modification statement (and any cascading actions and constraint checks) have completed successfully. The *INSTEAD OF* trigger is similar to the ANSI *BEFORE* trigger in that the code of the trigger may completely replace the data-modification operation. It specifies that the trigger is executed instead of the data-modification statement that fired the trigger. *INSTEAD OF DELETE* triggers cannot be used when there is a cascading action on the delete. Only *INSTEAD OF* triggers can access *TEXT*, *NTEXT*, or *IMAGE* columns.

```
WITH APPEND
```

Adds an additional trigger of an existing type to a table or view. This clause is supported for backward compatibility with earlier versions of the product and can be used only with *FOR* triggers.

```
NOT FOR REPLICATION
```

Prevents data-manipulation operations invoked through SQL Server's built-in replication engine from firing the trigger.

IF UPDATE(*column*) [{AND | OR} UPDATE(*column*)][...]

Allows you to choose the specific columns that fire the trigger. A column-specific trigger fires only on *UPDATE* and *INSERT* operations, but not on *DELETE* operations. If an *UPDATE* or *INSERT* occurs on any columns not in the column list, the trigger will not fire.

SQL Server allows multiple triggers for a given data-manipulation operation on a table or view. Thus, three *UPDATE* triggers are possible on a single table. Multiple *AFTER* triggers are possible on a given table. Their specific order is undefined, though the first and last triggers can be explicitly declared using the *sp_settriggerorder* system stored procedure. Only one *INSTEAD OF* trigger is possible per *INSERT*, *UPDATE*, or *DELETE* statement on a given table.

In SQL Server, any combination of triggers is made possible in a single trigger definition statement by separating each option with a comma. (When doing so, the same code fires for each statement in the combination definition.)

SQL Server implicitly fires in the *FOR EACH STATEMENT* style of the ANSI standard.

SQL Server instantiates two important pseudotables when a trigger is fired: **deleted** and **inserted**. They are equivalent to the **before** and **after** pseudotables, respectively, described earlier in the ANSI [Rules at a Glance](#) section. These tables are identical in structure to the table on which the triggers are defined, except that they contain the old data before the data-modification statement fired (*deleted*) and the new values of the table after the data-modification statement fired (*inserted*).

The *AS IF UPDATE(column)* clause tests specifically for *INSERT* or *UPDATE* actions on a given column or columns, in the way the ANSI statement uses the *UPDATE(column)* syntax. Specify multiple columns by adding separate *UPDATE(column)* clauses after the first. Follow the *AS IF UPDATE(column)* clause with a Transact-SQL *BEGIN...END* block to allow the trigger to fire multiple Transact-SQL operations. This clause is functionally equivalent to the *IF...THEN...ELSE* operation.

In addition to intercepting data modification statements as shown in the ANSI SQL example, SQL Server allows you to perform other sorts of actions when a data modification operation occurs. In the following example, we've decided that the table **sales_archive_2002** is off limits and that anyone who attempts to alter data in these tables is to be notified of that restriction:

```
CREATE TRIGGER archive_trigger
ON sales_archive_2002
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10, 'No changes allowed to this table')
GO
```

SQL Server *does not* allow the following statements within the Transact-SQL *code_block* of a trigger: *ALTER*, *CREATE*, *DROP*, *DENY*, *GRANT*, *REVOKE*, *LOAD*, *RESTORE*, *RECONFIGURE*, or *TRUNCATE*. In addition, it does not allow any *DISK* statements or the *UPDATE STATISTICS* command.

SQL Server also allows triggers to fire recursively using the *recursive triggers* setting of the **sp_dboption** system stored procedure. Recursive triggers, by their own action, cause themselves to fire again. For example, if an *INSERT* trigger on table **T1** performs an *INSERT* operation on table **T1**, it might perform a recursive operation. Since recursive triggers can be dangerous, this functionality is disabled by default.

Similarly, SQL Server allows *nested triggers* up to 32 levels deep. If any one of the nested triggers performs a *ROLLBACK* operation, no further triggers execute. An example of nested triggers is a trigger on table **T1** firing an operation against table **T2**, which also has a trigger that fires an operation against table **T3**. The triggers cancel if an infinite loop is encountered. Nested triggers are enabled with the nested triggers setting of the system stored procedure **sp_configure**. If nested triggers are disabled, recursive triggers are disabled as well, despite the recursive triggers setting of **sp_dboption**.

In the following example, we want to reroute the user activity that occurs on the **people** table, especially update transactions, so that changes to records in the **people** table are instead written to the **people_reroute** table. (A more elaborate version of the **people** table is shown in [SQL Server](#) in the [CREATE/ALTER TABLE Statement](#) section.) Our update trigger will record any changes to columns 2, 3, or 4 of the **people** table and write it to the **people_reroute** table. The trigger will also record which user issued the update transaction and the time of the transaction:

```
CREATE TABLE people
(
    people_id CHAR(4),
    people_name VARCHAR(40),
    people_addr VARCHAR(40),
    city VARCHAR(20),
    state CHAR(2),
    zip CHAR(5),

```

```
phone CHAR(12),
sales_rep empid NOT NULL)
GO
CREATE TABLE people_reroute
(reroute_log_id UNIQUEIDENTIFIER DEFAULT NEWID( ),
reroute_log_type CHAR (3) NOT NULL,
reroute_people_id CHAR(4),
reroute_people_name VARCHAR(40),
reroute_people_addr VARCHAR(40),
reroute_city VARCHAR(20),
reroute_state CHAR(2),
reroute_zip CHAR(5),
reroute_phone CHAR(12),
reroute_sales_rep empidNOT NULL,
reroute_user sysname DEFAULT SUSER_SNAME( ),
reroute_changed datetime DEFAULT GETDATE( ) )
GO
CREATE TRIGGER update_person_data
ON people
FOR update AS
IF (COLUMNS_UPDATED(people_name)
OR COLUMNS_UPDATEEE(people_addr)
OR COLUMNS_UPDATED(city) )
BEGIN
-- Audit OLD record.
INSERT INTO people_reroute (reroute_log_type, reroute_people_id,
reroute_people_name, reroute_people_addr, reroute_city)
SELECT 'old', d.people_id, d.people_name, d.people_addr, d.city
FROM deleted AS d
-- Audit NEW record.
INSERT INTO people_reroute (reroute_log_type, reroute_people_id,
reroute_people_name, reroute_people_addr, reroute_city)
SELECT 'new', n.people_id, n.people_name, n.people_addr, n.city
FROM inserted AS n
END
GO
```

Note that SQL Server *CREATE* statements allow *deferred name resolution*, meaning that a command is processed even if it refers to a database object that does not yet exist in the database.

See Also

CREATE?ALTER FUNCTION/PROCEDURE
DELETE
DROP
INSERT
UPDATE

CREATE/ALTER TYPE Statement

The *CREATE TYPE* statement allows you to create a *user-defined type* (UDT); that is, a user-defined datatype or "class" in object-oriented terms. UDTs extend SQL capabilities into the realm of object-oriented programming by allowing inheritance and other object-oriented features. You can also create something called *typed tables* with the *CREATE TABLE* statement using a previously created type made with the *CREATE TYPE* statement. Typed tables are based on UDTs and are equivalent to "instantiated classes" from object-oriented programming.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with variations
SQL Server	Not supported

SQL2003 Syntax

```
CREATE TYPE type_name
[UNDER supertype_name]
[AS [new_udt_name] datatype [attribute] [...]]
  {[REFERENCES ARE [NOT] CHECKED
  [ON DELETE
  {[NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT] } ] |
  [DEFAULT value] |
  [COLLATE collation_name] } ]
  [ [NOT] INSTANTIABLE]
  [ [NOT] FINAL]
[REF IS SYTEM GENERATED |
  REF USING datatype
  [CAST {(SOURCE AS REF) | (REF AS SOURCE) WITH identifier} ] |
  REF new_udt_name [...] } ]
[CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH identifier]
[method_definition [...]]
```

The syntax below alters an existing user-defined datatype:

```
ALTER TYPE type_name {ADD ATTRIBUTE type_definition |
  DROP ATTRIBUTE type_name}
```

Keywords

{CREATE | ALTER} TYPE *type_name*

Creates a new type or alters an existing type of *type_name*.

UNDER *supertype_name*

Creates a subtype that is dependent upon a single, pre-existing, named supertype. (A UDT can be a supertype if it is defined as *NOT FINAL*.)

AS [*new_udt_name*] *datatype* [*attribute*] [,...]

Defines attributes of the type as if they were column declarations in a *CREATE TABLE* statement without constraints. You should either define the UDT attribute on an existing datatype, like *VARCHAR(10)*, or on another, previously created UDT or even on a user-defined domain. Defining a UDT using a predefined datatype (e.g., *CREATE TYPE my_type AS INT*) creates a *distinct type*, while a UDT defined with an attribute definition is a *structured type*. The allowable attributes for a structured type are below:

ON DELETE NO ACTION

Produces an error when the foreign key is violated (the default).

ON DELETE RESTRICT

A synonym for *NO ACTION*.

ON DELETE CASCADE

Sets the value of the referencing column to the value of the referenced column.

ON DELETE SET NULL

Sets the referencing column value to NULL.

ON DELETE SET DEFAULT value

Defines a default value for the UDT when no value is supplied by the user. Follows the rules of a *DEFAULT* in the [CREATE/ALTER TABLE Statement](#) section.

COLLATE collation_name

Assigns a collation — that is, a sort order — for the UDT. When omitted, the collation of the database where the UDT was created applies. Follows the rules of a *COLLATION* in the [CREATE/ALTER TABLE Statement](#) section.

[*NOT*] *INSTANTIABLE*

Defines the UDT such that it can be instantiated. *INSTANTIABLE* is required for typed tables, but not standard UDTs.

[*NOT*] *FINAL*

Required for all UDTs. *FINAL* means the UDT may have no subtypes. *NOT FINAL* means the UDT may have subtypes.

REF

Defines either a system-generated or user-generated reference specification — that is, a sort of unique identifier that acts as a pointer that another type may reference. By referencing a pre-existing type using its reference specification, new types may inherit properties of pre-existing types. There are three ways to tell the DBMS how the typed table's reference column gets its values (i.e., its reference specification):

new_udt_name [,...]

Declares the reference specification is provided by another pre-existing UDT called *new_udt_name*.

IS SYSTEM GENERATED

Declares the reference specification is system-generated. (Think of an automatically incrementing column.) This is the default when the *REF* clause is omitted.

USING *datatype* [CAST {(SOURCE AS REF) | (REF AS SOURCE) WITH *identifier*]

Declares that the user defines the reference specification. You do this by using a predefined datatype and optionally casting the value. You can *CAST (SOURCE AS REF) WITH identifier* to cast the value with the specified *datatype* to the reference type of the structured type. You can *CAST (REF AS SOURCE) WITH identifier* to cast the value for the structured type to the *datatype*. The *WITH* clause allows you to declare an additional identifier for the cast datatype.

CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH *identifier method_definition* [,...]

Defines one or more pre-existing methods for the UDT. A *method* is merely a specialized user-defined function and is created using the *CREATE METHOD* statement (see *CREATE FUNCTION*). The *method_definition* clause is not needed for structured types since their method(s) are implicitly created. The default characteristics of a method are *LANGUAGE SQL*, *PARAMETER STYLE SQL*, *NOT DETERMINISTIC*, *CONTAINS SQL*, and *RETURN NULL ON NULL INPUT*.

ADD ATTRIBUTE *type_definition*

Adds an additional attribute to an existing UDT, using the format described earlier under the *AS* clause. Available via the *ALTER TYPE* statement.

DROP ATTRIBUTE *type_name*

Drops an attribute from an existing UDT. Available via the *ALTER TYPE* statement.

Rules at a Glance

You are able to create user-defined types (UDTs) as a way to further ensure data integrity in your database and to ease the work involved in doing so. An important concept of UDTs is that they allow you to easily create *subtypes*. Subtypes are UDTs built upon other UDTs (the supertype). Subtypes inherit the characteristics of their supertypes. The UDT that subtypes are dependent on is called a parent type or *supertype*.

Assume, for example, that you want to define a general UDT for phone numbers called **phone_nbr**. You could then easily build new subtypes of *phone_nbr* called **home_phone**, **work_phone**, **cell_phone**, **pager_phone**, etc. Each of the subtypes could then inherit the general characteristics of the parent type, but then have characteristics of their own. In the following example, we create a general root UDT called **money** and then several subtypes:

```
CREATE TYPE money (phone_number DECIMAL (10,2) )
```

```
    NOT FINAL;
```

```
CREATE TYPE dollar UNDER money AS DECIMAL(10,2)
```

```
    (conversion_rate DECIMAL(10,2) ) NOT FINAL;
```

```
CREATE TYPE euro UNDER money AS DECIMAL(10,2)
```

```
    (dollar_conversion_rate DECIMAL(10,2) ) NOT FINAL;
```

```
CREATE TYPE pound UNDER euro
```

```
(euro_conversion_rate DECIMAL(10,2) ) FINAL;
```

Programming Tips and Gotchas

The biggest programming gotcha for user-defined types is that they are simply seldom used or well understood by most database developers and database administrators. Consequently, they can be problematic due to simple ignorance. They offer, however, a consistent and labor-saving approach for representing commonly reused conceptual elements in a database, such as address (e.g., **street1**, **street2**, **city**, **state**, **postal code**).

DB2

DB2 supports most elements of the ANSI standard with a few specific enhancements. The *CREATE TYPE* syntax is:

```
CREATE [DISTINCT] TYPE type_name
[UNDER supertype_name]
[AS [new_udt_name] datatype]
[WITH COMPARISONS]
[[NOT] INSTANTIABLE] [INLINE LENGTH int] [WITHOUT COMPARISONS]
[NOT FINAL] MODE DB2SQL
[REF USING datatype] [CAST {(SOURCE AS REF) | (REF AS SOURCE)}
  WITH identifier } ]
[[OVERRIDING] METHOD method_definition [,... ] ]
```

The syntax below alters an existing user-defined datatype on DB2:

```
ALTER TYPE type_name {ADD ATTRIBUTE type_definition |
  DROP ATTRIBUTE type_name [RESTRICT] |
  ADD METHOD method_definition |
  ALTER METHOD method_name [[NOT] FENCED] [[NOT] THREADSAFE] |
  DROP METHOD method_name [RESTRICT] }
```

The parameters are:

DISTINCT

Defines a distinct type. A distinct type is always sourced from one of the existing datatypes that ships with DB2. *CREATE DISTINCT TYPE* generates one or more function to cast between the distinct type and its source type.

WITH COMPARISONS

Generates support for the comparison operators (=, <>, <, <=, >, and >=) for use with a distinct type. This clause is used only with *CREATE DISTINCT TYPE*. Even when creating a distinct type, *WITH COMPARISONS* should not be used if the source *datatype* is *BLOB*, *CLOB*, *DBCLOB*, *LONG VARCHAR*, *LONG VARGRAPHIC*, or *DATALINK*.

INLINE LENGTH *int*

Defines the maximum length in bytes, *int*, of a structured type column. When omitted, DB2 calculates the maximum length itself.

WITHOUT COMPARISONS

Declares that there are no comparison functions supported for the structure type.

MODE DB2SQL

This is a required clause needed to maintain ANSI compliance.

OVERRIDING

Declares that the method of a subtype overrides a method of the same name on the supertype. This allows easy reimplementations of methods in subtypes.

METHOD *method_definition* [...]

DB2 supports a number of SQL routine characteristics and external routine characteristics on the *method_definition*. In short, you must define a method in the *CREATE/ALTER TYPE* statement, but then you must also define the implementation of the method using *CREATE METHOD*. Creating methods are explained in [CREATE/ALTER FUNCTION/PROCEDURE Statements](#).

ADD METHOD *method_definition*

Adds a new method, *method_definition*, to the existing type, *type_name*. However, the method is not usable until a *CREATE METHOD* statement defines the implementation of the method.

ALTER METHOD *method_name*

Alters a method, but the method must not be *LANGUAGE SQL*.

[NOT] FENCED

Declares whether the method is safe to run in the database's operating system process (*NOT FENCED*) or not (*FENCED*). *NOT FENCED* methods can introduce integrity problems when not adequately set up to prevent errors. This clause is incompatible with *NOT THREADSAFE*.

[NOT] THREADSAFE

Declares whether the method is safe to run in the same process as other routines (*THREADSAFE*) or not (*NOT THREADSAFE*).

DROP METHOD *method_name*

Drops an existing method of *method_name*.

RESTRICT

Disallows an attribute or method from being dropped when the type, *type_name*, is used in other existing objects, such as a table, view, column, index extension, etc.

In this example, we create a supertype for **employee** and a subtype for **manager**:

```
CREATE TYPE employee AS
  (last_name VARCHAR(32),
   emp_id INT,
   dept REF(dept),
   salary DECIMAL(10,2))
```

```
MODE DB2SQL;  
  
CREATE TYPE manager UNDER emp AS  
    (bonus DECIMAL(10,2))  
  
MODE DB2SQL;
```

The following example shows a more elaborate type hierarchy for addresses:

```
CREATE TYPE address_type AS  
    (street1  VARCHAR(30),  
     street2  VARCHAR(30),  
     city     VARCHAR(30),  
     state    VARCHAR(10),  
     locality_code CHAR(15) )  
  
NOT FINAL  
  
MODE DB2SQL  
  
    METHOD SAMEZIP (addr address_type)  
  
    RETURNS INTEGER  
  
    LANGUAGE SQL  
  
    DETERMINISTIC  
  
    CONTAINS SQL  
  
    NO EXTERNAL ACTION  
  
    METHOD DISTANCE (address_type)  
  
    RETURNS FLOAT  
  
    LANGUAGE C  
  
    DETERMINISTIC  
  
    PARAMETER STYLE SQL  
  
    NO SQL  
  
    NO EXTERNAL ACTION  
  
CREATE TYPE spain_address_type UNDER address_type AS  
    (family_name VARCHAR(30))  
  
NOT FINAL  
  
MODE DB2SQL  
  
CREATE TYPE usa_address_type UNDER address_type AS  
    (zip VARCHAR(10))  
  
NOT FINAL  
  
MODE DB2SQL
```

In the above example, we created a type called **address_type** with two subtypes: **spain_address_type** and **usa_address_type**.

The example that follows shows how to create a distinct type in DB2 used to identify an employee mailstop.


```
CREATE DISTINCT TYPE mailstop AS CHAR(5)

WITH COMPARISONS;
```

In essence, the type **mailstop** now functions like a regular datatype and may be used to create columns in tables, variables in stored procedures, etc. The *WITH COMPARISONS* clause tells DB2 to create the standard pre-existing system operators for use with the distinct type. Specific functions, such as *AVG*, *MIN*, or *LENGTH*, may not work unless the *CREATE FUNCTION* statement is used to register the function for the distinct type.

MySQL

Not supported.

Oracle

Oracle has *CREATE TYPE* and *ALTER TYPE* statements, but they are not ANSI standard. Instead of a single *CREATE TYPE* statement, Oracle uses *CREATE TYPE BODY* to define the code that makes up the UDT, while *CREATE TYPE* defines the argument specification for the type. The syntax for *CREATE TYPE* is:

```
CREATE [OR REPLACE] TYPE type_name

[AUTHID {DEFINER | CURRENT_USER} ]

{ [ UNDER supertype_name ] |

  {AS | IS} [OID 'object_identifier' ]

  {OBJECT | TABLE OF datatype | VARRAY ( limit ) OF datatype} ] }

  EXTERNAL NAME java_ext_name LANGUAGE JAVA USING data_definition

  { [ ( attribute datatype [,...] [EXTERNAL NAME 'name' ] |

    [[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]

    [{ {MEMBER | STATIC}

      {function_based | procedure_based} | constructor_clause |

      map_clause } [...] ]

    [pragma_clause] ] }
```

Once the type has been declared, you encapsulate all of the UDT logic in the type body declaration. The *type_name* for both the *CREATE TYPE* and *CREATE TYPE BODY* should be identical. The syntax for the type body is shown below:

```
CREATE [OR REPLACE] TYPE BODY type_name

{AS | IS}

{{MEMBER | STATIC}

{function_based | procedure_based | constructor_clause}}[...]

[map_clause]
```

Oracle's implementation of *ALTER TYPE* enables you to drop or add attributes and methods from the type:

```
ALTER TYPE type_name

{COMPILE [DEBUG] [{SPECIFICATION | BODY}]

  [compiler_directives] [REUSE SETTINGS] |

  REPLACE [AUTHID {DEFINER | CURRENT_USER}] AS OBJECT

  ( attribute datatype [,...] [element_definition [,...] ] ) |
```

```
[[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]
{ {ADD | DROP} { {MAP | ORDER} MEMBER FUNCTION function_name
  (parameter datatype [,...]) } |
  {MEMBER | STATIC} {function_based | procedure_based} |
  constructor_clause | map_clause } [...] |
  {ADD | DROP | MODIFY} ATTRIBUTE (attribute [datatype] [,...]) |
  MODIFY {LIMIT int | ELEMENT TYPE datatype} }
[ {INVALIDATE |
  CASCADE [{ [NOT] INCLUDING TABLE DATA | CONVERT TO SUBSTITUTABLE}]
  [FORCE] [EXCEPTIONS INTO table_name] } ]
```

Parameters for the three statements are as follows:

OR REPLACE

Recreates the UDT if it already exists. Objects that depend on the type are marked as *DISABLE* after you recreate the type.

AUTHID {DEFINER | CURRENT_USER}

Determines what user permissions any member functions or procedures are executed under and how external name references are resolved. (Note that subtypes inherit the permission style of their supertype.) This clause can only be used with an *OBJECT* type, not with a *VARRAY* type or nested table type, according to the following keywords:

DEFINER

Executes functions or procedures under the privileges of the user who created the UDT. Also specifies that unqualified object names (object names without a schema definition) in SQL statements are resolved to the schema where the member functions or procedures reside.

CURRENT_USER

Executes functions or procedures under the privileges of the user who invoked the UDT. Also specifies that unqualified object names in SQL statements are resolved to the schema of the user who invoked the UDT.

UNDER *supertype_name*

Declares that the UDT is a subtype of another, pre-existing UDT. The supertype UDT must be created with the *AS OBJECT* clause. A subtype will inherit the properties of the supertype, though you should override or add new properties to differentiate it from the supertype.

OID '*object_identifier*'

Declares an equivalent identical object, of name '*object_identifier*', in more than one database. This clause is most commonly used by those developing Oracle Data Cartridges and is seldom used in standard SQL statement development.

AS OBJECT

Creates the UDT as a root object type, the top level object in a UDT hierarchy of objects.

AS TABLE OF *datatype*

Creates a named nested table type of a UDT called *datatype*. The datatype cannot be an *NCLOB*, but *CLOB* or *BLOB* are acceptable. If the datatype is an object type, then the columns of the nested table must match the name and attributes of the object type.

AS VARRAY (*limit*) OF *datatype*

Creates the UDT as an ordered set of elements, all of the same datatype. The *limit* is an integer of zero or more. The type name must be a built-in datatype, a *REF*, or an object type. The *VARRAY* cannot contain *LOB* or *XMLType* datatypes.

EXTERNAL NAME *java_ext_name* LANGUAGE JAVA USING *data_definition*

Maps a Java class to a SQL UDT by specifying the name of a public Java external class, *java_ext_name*. Once defined, all objects in a Java class must be Java objects. The *data_definition* may be *SQLData*, *CustomDatum*, or *OraData* as defined in the "Oracle9i JDBC Developers Guide." You can map many Java object types to the same class, but there are two restrictions. First, do not map to the same class two or more subtypes of a common datatype. Second, subtypes must be mapped to an immediate subclass of the class that their supertype is mapped to.

datatype

Declares the attributes and datatypes used by the UDT. Oracle does not allow *ROWID*, *LONG*, *LONG ROW*, or *UROWID*. Nested tables and varrays do not allow attributes of **AnyType**, **AnyData**, or **AnyDataSet**.

EXTERNAL NAME '*name*' [NOT] OVERRIDING

Declares that this method overrides a *MEMBER* method defined in the supertype (*OVERRIDING*) or not (*NOT OVERRIDING*), the default. This clause is valid only for *MEMBER* clauses.

MEMBER | *STATIC*

Describes the way in which subprograms are associated with the UDT as attributes. *MEMBER* has an implicit first argument referenced as *SELF*, as in *object_expression.method()*. *STATIC* has no implicit arguments, as in *type_name.method()*. The following basis are allowed to call subprograms:

function_based

Declares a subprogram that is function-based using the syntax:

```
FUNCTION function_name ( parameter datatype [...] )
```

```
return_clause | java_object_clause
```

The *function_name* cannot be the name of any existing attribute, include those inherited from supertypes. This clause allows you to define the PL/SQL function-based UDT body without resorting to the *CREATE TYPE BODY* statement. The *return_clause* and *java_object_clause* are defined in the following text.

procedure_based

Declares a subprogram that is function-based using the syntax:

```
PROCEDURE procedure_name ( parameter datatype [...] )
```

```
{AS | IS} LANGUAGE {java_call_spec | c_call_spec}
```

Refer to the entry on *java* and *c_call_spec* later in this list. This clause allows you to define the PL/SQL procedure-based UDT body without resorting to the *CREATE TYPE BODY* statement. The *procedure_name* cannot be the name of any existing attribute, include those inherited from supertypes.

constructor_clause

Declares one or more constructor specifications using the syntax:

```
[FINAL] [INSTANTIABLE] CONSTRUCTOR FUNCTION datatype  
  [ ( [SELF IN OUT datatype ,] {parameter datatype [,...]} ) ]  
RETURN SELF AS RESULT  
  [ {AS | IS} LANGUAGE {java_call_spec | c_call_spec}
```

A constructor specification is a function that returns an initialized instance of a UDT. Constructor specifications are always *FINAL*, *INSTANTIABLE*, and *SELF IN OUT*, so these keywords are not required. The *java_call_spec* and *c_call_spec* subclauses may be replaced with a PL/SQL code block in the *CREATE TYPE BODY* statement. (Refer to the entries on *java_call_spec* and *c_call_spec* later in this list.)

map_clause

Declares the mapping or ordering of a supertype using the syntax:

```
{MAP | ORDER} MEMBER function_based
```

Refer to the entry on *function_based* earlier in this list. *MAP* uses more efficient algorithms for object comparison and is best in situations when performing extensive sorting or hash joins. *MAP MEMBER* specifies the relative position of a given instance in the ordering of all instances of the UDT. *ORDER MEMBER* specifies the explicit position of an instance and references a *function_based* subprogram that returns a NUMBER datatype value.

return_clause

Declares the datatype return format of the SQL UDT using the syntax:

```
RETURN datatype [ {AS | IS} LANGUAGE {java_call_spec |  
c_call_spec}.
```

java_object_clause

Declares the return format of the Java UDT using the syntax:

```
RETURN {datatype | SELF AS RESULT} EXTERNAL [VARIABLE] NAME  
  'java_name'
```

If you use the *EXTERNAL* clause, the value of the public Java method must be compatible with the SQL returned value.

pragma_clause

Declares a *pragma restriction* (that is, an Oracle precompiler directive) for the type using the syntax:

```
PRAGMA RESTRICT REFERENCES ( {DEFAULT | method_name},  
{RNDS | WNDS |RNPS | WNPS | TRUST}{,...} )
```

This feature is deprecated and *should be avoided*. It is intended to control how UDTs read and write database tables and variables. The parameters in the *PRAGMA* syntax are:

DEFAULT

Applies the pragma to all methods in the type that don't have another pragma in place.

method_name

Identifies the exact method to which to apply the pragma.

RNDS

Reads no database state — no database reads allowed.

WNDS

Writes no database state — no database writes allowed.

RNPS

Reads no package state — no package reads allowed.

WNPS

Writes no package state — no package writes allowed.

TRUST

States that the restrictions of the pragma are assumed but not enforced.

java_call_spec

Identifies the Java implementation of a method using the syntax *JAVA NAME 'string'*. This clause allows you to define the Java UDT body without resorting to the *CREATE TYPE BODY* statement.

c_call_spec

Declares a C language call specification using the syntax:

C [NAME *name*] LIBRARY *lib_name* [AGENT IN (*argument*)]
[WITH CONTEXT] [PARAMETERS (*parameter* [,...])]

This clause allows you to define the C UDT body without resorting to the *CREATE TYPE BODY* statement.

COMPILE

Compiles the object type specification and body. This is the default when neither a *SPECIFICATION* clause nor a *BODY* clause are defined.

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive = 'value'*. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

DEBUG

Generates and stores additional codes for the PL/SQL debugger. Do not specify both *DEBUG* and the *compiler_directive* *PLSQL_DEBUG*.

SPECIFICATION | *BODY*

Chooses whether to recompile the *SPECIFICATION* of the object type (created by the *CREATE TYPE* statement) or the *BODY* (created by the *CREATE TYPE BODY* statement).

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive* = '*value*'. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

REUSE SETTINGS

Retains the original value for the *compiler_directives* mentioned earlier.

REPLACE AS OBJECT

Adds new member subtypes to the specification. This clause is valid only for object types.

[NOT] OVERRIDING

Indicates the method overrides a *MEMBER* method defined in the supertype. This clause is only valid with *MEMBER* methods and is required for methods that define (or redefine using *ALTER*) a supertype method. *NOT OVERRIDING* is the default if this clause is omitted.

ADD

Adds a new *MEMBER* function, function- or procedure-based subprogram, or attribute to the UDT.

DROP

Drops an existing *MEMBER* function, function- or procedure-based subprogram, or attribute from the UDT.

MODIFY

Alters the properties of an existing attribute of the UDT.

MODIFY LIMIT int

Increases the number of elements in a *VARRAY* collection type up to *int*, as long as *int* is greater than the current number of elements in the *VARRAY*. It is not valid for nested tables.

MODIFY ELEMENT TYPE datatype

Increases the precision, size, or length of a scalar datatype of a *VARRAY* or nested table. This clause is valid for any non-object collection type. If the collection is a *NUMBER*, you may increase its precision or scale. If the collection is *RAW*, you may increase its maximum size. If the collection is *VARCHAR2* or *NVARCHAR2*, you may increase its maximum length.

INVALIDATE

Invalidates all dependent objects without checks.

CASCADE

Cascades the change to all subtypes and tables. By default, the action will be rolled back if any errors are encountered in the dependent types or tables.

[NOT] INCLUDING TABLE DATA

Converts data stored in the UDT columns to the most recent version of the column's type (*INCLUDING TABLE DATA*), the default, or not (*NOT INCLUDING TABLE DATA*). When *NOT*, Oracle checks the metadata, but does not check or update the dependent table data.

CONVERT TO SUBSTITUTABLE

Used when changing a type from *FINAL* to *NOT FINAL*. The altered type then can be used in substitutable tables and columns, as well as in subtypes, instances of dependent tables, and columns.

FORCE

Causes the *CASCADE* operation to go forward, ignoring any errors found in dependent subtypes and tables. All errors are logged to a previously created *EXCEPTIONS* table.

EXCEPTIONS INTO *table_name*

Logs all error data to a table previously created using the system package called `DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE`.

In this example, we create a Java SQLJ object type called **address_type**:

```
CREATE TYPE address_type AS OBJECT
  EXTERNAL NAME 'scott.address' LANGUAGE JAVA
  USING SQLDATA ( street1 VARCHAR(30) EXTERNAL NAME 'str1',
    street2      VARCHAR(30) EXTERNAL NAME 'str2',
    city         VARCHAR(30) EXTERNAL NAME 'city',
    state        CHAR(2)   EXTERNAL NAME 'st',
    locality_code CHAR(15) EXTERNAL NAME 'lc',
    STATIC FUNCTION square_feet RETURN NUMBER
      EXTERNAL VARIABLE NAME 'square_feet',
    STATIC FUNCTION create_addr (str VARCHAR,
      City VARCHAR, state VARCHAR, zip NUMBER)
      RETURN address_type
      EXTERNAL NAME 'create (java.lang.String,
        java.lang.String, java.lang.String, int)
        return scott.address',
    MEMBER FUNCTION rtrim_spaces RETURN SELF AS RESULT
      EXTERNAL NAME 'rtrim_spaces ( ) return scott.address' )
NOT FINAL;
```

We could create a UDT using a *VARRAY* type with four elements:

```
CREATE TYPE employee_phone_numbers AS VARRAY(4) OF CHAR(14);
```

In the following example, we alter the **address_type** that we created earlier by adding a *VARRAY* called **phone_varray**:

```
ALTER TYPE address_type
  ADD ATTRIBUTE (phone phone_varray) CASCADE;
```

In this last example, we'll create a supertype and subtype called **menu_item_type** and **entry_type**, respectively:

```
CREATE OR REPLACE TYPE menu_item_type AS OBJECT
(id INTEGER, title VARCHAR2(500),
  NOT INSTANTIABLE
  MEMBER FUNCTION fresh_today
  RETURN BOOLEAN)
NOT INSTANTIABLE
NOT FINAL;
```

In the above example, we created a type specification (but not the type body) that defines items that may appear on a menu at a café. Included with the type specification is a subprogram method called **fresh_today**, a Boolean indicator that tells whether the menu item is made fresh that day. The *NOT FINAL* clause that appears at the end of the code tells Oracle that this type may serve as the supertype (or base type) to other subtypes that we might derive from it. So now, let's create the **entre_type**:

```
CREATE OR REPLACE TYPE entry_type UNDER menu_item_type
(entre_id INTEGER, desc_of_entre VARCHAR2(500),
  OVERRIDING MEMBER FUNCTION fresh_today
  RETURN BOOLEAN)

NOT FINAL;
```

PostgreSQL

PostgreSQL supports a *CREATE TYPE* statement used to create a new datatype, but does not support an *ALTER TYPE* statement. PostgreSQL's implementation of the *CREATE TYPE* statement is nonstandard:

```
CREATE TYPE type_name
( INPUT = input_function_name,
  OUTPUT = output_function_name,
  INTERNALLENGTH = { int | VARIABLE }
[, DEFAULT = value]
[, ELEMENT = array_element_datatype]
[, DELIMITER = delimiter_character]
[, PASSEDBYVALUE ]
[, ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE} ]
[, STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN} ] )
```

The parameters are as follows:

```
CREATE TYPE type_name
```

Creates a new user-defined datatype called *type_name*. The name may not exceed 30 characters in length nor begin with an underscore.

INPUT = *input_function_name*

Declares the name of a previously created function that converts external argument values into a form usable by the type's internal form.

OUTPUT = *output_function_name*

Declares the name of a previously created function that converts internal output values to a display format.

INTERNALLENGTH = { *int* | VARIABLE }

Specifies a numeric value, *int*, for the internal length of the new type, if the datatype is fixed length. The keyword *VARIABLE* declares that the internal length is variable. The default is *VARIABLE*.

DEFAULT = *value*

Defines a value for *type* when it defaults.

ELEMENT = *array_element_datatype*

Declares that the datatype is an array and that *array_element_datatype* is the datatype of the array elements. For example, an array containing integer values would be *ELEMENT = INT4*. In general, you should allow the *array_element_datatype* value to default. The only time you might want to override the default is when creating a fixed-length UDT composed of an array of multiple identical elements that need to be directly accessible by subscripting.

DELIMITER = *delimiter_character*

Declares a character used as a delimiter between output values of an array produced by the type. Only used with the *ELEMENT* clause. The default is a comma.

PASSEDBYVALUE

Specifies that the values of the datatype are passed by value and not by reference. The optional clause cannot be used for types whose value is longer than the *DATUM* datatype (4 bytes on most operating systems, 8 bytes on a few others).

ALIGNMENT = { *CHAR* | *INT2* | *INT4* | *DOUBLE* }

Defines a storage alignment for the type. Four datatypes are allowed, with each one equating to a specific boundary. *CHAR* equals a 1-byte boundary. *INT2* equals a 2-byte boundary. *INT4* equals a 4-byte boundary (the requirement for a variable-length UDT on PostgreSQL). *DOUBLE* equals an 8-byte boundary.

STORAGE = { *PLAIN* | *EXTERNAL* | *EXTENDED* | *MAIN* }

Defines a storage technique for variable-length UDTs. (*PLAIN* is required for fixed-length UDTs.) Four types are allowed:

PLAIN

Stores the UDT in-line, when declared as the datatype of a column in a table, and uncompressed.

EXTERNAL

Stores the UDT outside of the table without trying to compress it first.

EXTENDED

Stores the UDT as a compressed value if it fits inside of the table. If it is still too long, PostgreSQL will save the UDT outside of the table.

MAIN

Stores the UDT as a compressed value within the table. There are situations where PostgreSQL cannot save the UDT within the table because it is just too large. The *MAIN* storage parameter puts the highest emphasis on storing UDTs with all other table data.

When you create a new datatype in PostgreSQL, it is available *only* in the current database. The user who created the datatype is the owner.

In order to create a new datatype, you must create at least two functions before defining the type (see [CREATE/ALTER FUNCTION/PROCEDURE Statements](#)). In summary, you must create an *INPUT* function that provides the type with external values that can be used by the operators and functions defined for the type. You must also create an *OUTPUT* function that renders a usable external representation of the datatype. There are some additional requirements when creating the input and output functions:

- The input function should either take one argument of type OPAQUE or take three arguments of OPAQUE, OID, and INT4. In the latter case, OPAQUE is the input text of a C string, OID is the element type for array types, and the INT4 (if known) is the typemod of the destination column.
- The output function should either take one argument of type OPAQUE or take two arguments of OPAQUE and OID. In the latter case, OPAQUE is the datatype itself and the OID is the element type for array types, if needed.

For example, we can create a UDT called **floorplan** and use it to define a column in two tables, one called **house** and one called **condo**:

```
CREATE TYPE floorplan
```

```
(INTERNALLENGTH=12, INPUT=squarefoot_calc_proc,
```

```
OUTPUT=out_floorplan_proc);
```

```
CREATE TABLE house
```

```
(house_plan_id int4,
```

```
size floorplan,
```

```
descrip varchar(30) );
```

```
CREATE TABLE condo
```

```
(condo_plan_id INT4,
```

```
size floorplan,
```

```
descrip varchar(30)
```

```
location_id varchar(7) );
```

SQL Server

Not supported. New datatypes can be added to SQL Server using the non-ANSI system stored procedure **sp_addtype**.

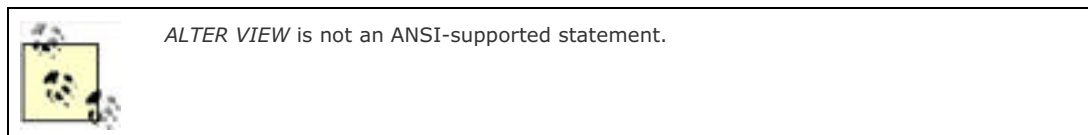
See Also

```
CREATE/ALTER FUNCTION/PROCEDURE  
DROP
```

CREATE/ALTER VIEW Statement

This statement creates a *view*, also known as a *virtual table*. A view acts just like a table but is actually defined as a query. When a view is referenced in a statement, the result set of the query becomes the content of the view for the duration of that statement. Almost any valid *SELECT* statement can define the contents of a view, though some platforms restrict the certain clauses of the *SELECT* statement and certain set operators.

In some cases, views can be updated, causing the view changes to be translated to the underlying data in the base tables. Some database platforms support a *materialized view*; that is, a physically created table that is defined with a query just like a view.



Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE [RECURSIVE] VIEW view_name {[column [,...]]} |  
[OF udt_name [UNDER supertype_name  
  [REF IS column_name {SYSTEM GENERATED | USER GENERATED | DERIVED} ]  
  [column_name WITH OPTIONS SCOPE table_name] ] }  
AS  
select_statement [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Keywords

CREATE VIEW *view_name*

Creates a new view using the supplied name.

RECURSIVE

Creates a view that derives values from itself. It must have a *column* clause and may not use the *WITH* clause.

[(*column* [,...])]

Names all of the columns in the view. The number of columns declared here must match the number of columns generated by the *select_statement*. When omitted, the columns assume the names of the columns that they come from. This clause is required when one or more of the columns is derived and does not have a base table column to reference.

OF *udt_name* [UNDER *supertype_name*]

Defines the view on a UDT rather than on the *column* clause. The typed view is created using each attribute of the type as a column in the view. Use the *UNDER* clause to define a view on a subtype.

REF IS *column_name* {SYSTEM GENERATED | USER GENERATED | DERIVED}

Defines the object-ID column for the view.

column_name WITH OPTIONS SCOPE *table_name*

Provides scoping for a reference column in the view. (Since the columns are derived from the type, there is no column list. Therefore, to specify column options, you must use "*column_name* WITH OPTIONS...".)

AS *select_statement*

Defines the exact *SELECT* statement that provides the data of the view.

[WITH [CASCADED | LOCAL] CHECK OPTION]

Used *only* on views that allow updates to their base table. It ensures that only data that may be read by the view may be inserted, updated, or deleted by the view. For example, if a view of **employees** showed salaried employees but not hourly employees, it would be impossible to insert, update, or delete hourly employees through that view. The *CASCADED* and *LOCAL* options of the *CHECK OPTION* clause are used for nested views. *CASCADED* performs the check option for the current view and all views it is built on top of. *LOCAL* performs the check option only for the current view, even when it is built upon other views.

Rules at a Glance

Views are usually as effective as the queries upon which they are based. That is why it is important to be sure that the defining *SELECT* statement is speedy and well-written. The simplest view is based on the entire contents of a single table:

```
CREATE VIEW employees
```

```
AS
```

```
SELECT *
```

```
FROM employee_tbl;
```

A column list also may be specified after the view name. The optional column list contains aliases serving as names for each element in the result set of the *SELECT* statement. If you use a column list, then you must provide a name for every column returned by the *SELECT* statement. If you don't use a column list, then the columns of the view will be named whatever the columns in the *SELECT* statement are called. You will sometimes see complex *SELECT* statements within a view that make heavy use of *AS* clauses for all columns, because that allows the developer of the view to put meaningful names on the columns without including a column list.

The ANSI standard defines that you must use a column list or an *AS* clause. However, some vendors allow more flexibility, so follow these rules for when to use an *AS* clause:

- When the *SELECT* statement contains calculated columns, such as (*salary * 1.04*)
- When the *SELECT* statement contains fully qualified column names, such as **pubs.scott.employee**
- When the *SELECT* statement contains more than one column of the same name (though with separate schema or database prefixes)

For example, the following two view declarations have the same functional result:

```
-- using a column-list
```

```
CREATE VIEW title_and_authors
```

```
(title, author_order, author, price, avg_monthly_sales,
```

```
publisher)
AS
SELECT t.title, ta.au_ord, a.au_lname, t.price, (t.ytd_sales / 12),
       t.pub_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;
-- using the AS clause with each column
CREATE VIEW title_and_authors
AS
SELECT t.title AS title, ta.au_ord AS author_order,
       a.au_lname AS author, t.price AS price,
       (t.ytd_sales / 12) AS avg_monthly_sales, t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0
```

You can alternately change the title of columns using the column list. In this case, we'll change **avg_monthly_sales** to **avg_sales**. Note that the code overrides the default column names provided by the AS clauses (in bold):

```
CREATE VIEW title_and_authors
(title, author_order, author, price, avg_sales, publisher)
AS
SELECT t.title AS title, ta.au_ord AS author_order,
       a.au_lname AS author, t.price AS price,
       (t.ytd_sales / 12) AS avg_monthly_sales, t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;
```

ANSI standard view can update the base table(s) it is based upon if it meets the following conditions:

- The view does not have *UNION*, *EXCEPT*, or *INTERSECT* operators.
- The defining *SELECT* statement does not contain *GROUP BY* or *HAVING* clauses.
- The defining *SELECT* statement does not contain any reference to non-ANSI pseudo-columns such as *ROWNUM* or *ROWGUIDCOL*.
- The defining *SELECT* statement does not contain the *DISTINCT* clause.

- The view is not materialized.

This example shows a view named **california_authors** that allows data modifications to apply only to authors within the state of California:

```
CREATE VIEW california_authors
```

```
AS
```

```
SELECT au_lname, au_fname, city, state
```

```
FROM authors
```

```
WHERE state = 'CA'
```

```
WITH LOCAL CHECK OPTION
```

The view shown in this example would accept *INSERT*, *DELETE*, and *UPDATE* statements against the base table, but still maintain that all inserted, updated, or deleted records contain a **state** of 'CA' using the *WITH ... CHECK* clause.

The most important rule to remember when updating a base table through a view is that all columns in a table that are defined as *NOT NULL* must receive a not null value when receiving a new or changed value. You can do this explicitly by directly inserting or updating a not null value into the column or by relying on a default value. In addition, views do not lift constraints on the base table. Thus, the values being inserted or updated into the base table must meet all the constraints placed on it by unique indexes, primary keys, *CHECK* constraints, etc.

Programming Tips and Gotchas

Views also can be built upon other views, but this is inadvisable and usually considered bad practice. Depending on the platform, such a view may take longer to compile, but may offer the same performance as a transaction against the base table(s). On other platforms, where each view is dynamically created as it is invoked, nested views may take a long time to return a result set because each level of nesting means another query must be processed before a result set is returned to the user. So in this worst case scenario, a three-level nested view must make three correlated query calls before it can return results to the user.

Although materialized views are defined like views, they take up space more like tables. Ensure that you have enough space available for the creation of materialized views.

DB2

DB2 supports a number of extensions to the ANSI standard *CREATE VIEW* statement:

```
CREATE VIEW view_name
```

```
{ [ (column [,...]) ] | [OF type_name MODE DB2SQL
```

```
  { (REF IS objectid_column_name USER GENERATED
```

```
    [UNCHECKED] [, with_options] ) |
```

```
  UNDER parent_view_name INHERIT SELECT PRIVILEGES
```

```
  [ (with_options) ] }
```

```
AS
```

```
[WITH common_table_expression [, ...] ]
```

```
(select_statement)
```

```
[WITH {CASCADE | LOCAL} CHECK OPTION]
```

The *ALTER VIEW* syntax is:

```
ALTER VIEW view_name
```

```
ALTER [COLUMN] column_name ADD SCOPE type_name
```

Parameters for both statements are as follows:

OF *type_name* MODE DB2SQL

Declares that the columns of the view are derived from attributes of a structured type called *type_name*. *MODE DB2SQL* indicates the type is DB2's version of a typed view.

REF IS *objectid_column_name* USER GENERATED

Declares that the object identifier column appears as the first column in the view. That first column in the *SELECT* clause of the view must be a *REF* type. In this case, the *objectID* is required on the topmost (i.e., root) view of a view hierarchy. The root view must be a typed view containing the *OF* clause that is not a subview. The *USER GENERATED* clause specifies that the first value of the *objectID* must be provided by the user. From then on, the value is maintained by the system.

UNCHECKED

Declares that the user is certain that all values in first column of the *SELECT* clause are unique and are not null. Uniqueness may be enforced by a primary key, unique key, unique index, *OID-column* definition, or an external method. This setting is usually optional, but is required for views in a typed hierarchy.

UNDER *parent_view_name* INHERIT SELECT PRIVILEGES

Declares that the view is a subview of the previously declared view *parent_view_name*. The child and parent views must be in the same schema. The parent view must have been declared with the *REF* clause and must be sourced from a structured type. The parent view named cannot have any subviews of the same structured type as the view being created. The *INHERIT SELECT PRIVILEGES* clause entitles any user or role with *SELECT* privileges on the parent view to also have *SELECT* privileges on all subviews.

with_options

Declares additional options for specific columns of the view. Multiple options are allowed by placing a comma between each option. The full syntax of the *with_options* clause is:

{*column_name* WITH OPTIONS {*READ ONLY* | *SCOPE type_name*} [,...]} [,...]

The parameters are:

READ ONLY

Specifies that a column is read-only.

SCOPE type_name

Specifies the scope of a reference type column where *type_name* is the name of the typed table or view. A scope is required for any column serving as the left operand of a dereference operator or as the argument of the *DEREF* function.

WITH *common_table_expression*

Associates a common table expression with the *SELECT* statement of the view. This clause is incompatible with typed views.

WITH {*CASCADE* | *LOCAL*} *CHECK OPTION*

Declares that any rows inserted or updated through the view must conform to the definition of the view. Any row that couldn't also be returned by the view cannot, therefore, be inserted or updated through it. When omitted, no checking of inserted or updated data is done beyond the constraints that exist on the source table. *WITH CHECK OPTION* is not allowed for read-only views, views with an *INSTEAD OF* trigger, or views that reference a nicknamed source, *NODENUMBER* or *PARTITION* functions, non-deterministic functions, or functions with external actions. The following *WITH CHECK OPTION* options are available:

CASCADE

Causes the search conditions of the current view to inherit the search conditions of any updateable view that the current view is based on. With this option, the current view has its own *WHERE* clause implicitly appended with an *AND* keyword to the *WHERE* clause of the view it depends on.

LOCAL

Applies the search conditions of the current view to any views that depend on it.

ALTER [COLUMN] *column_name* ADD SCOPE *type_name*

Allows you to add scope characteristics to an existing view.

The *SELECT* statement of a view should be a standard *SELECT* statement and cannot contain any of the following:

- Temporary tables
- Parameter markers
- Referenced host variables

You can get around the limitation on parameterized views by making a user-defined function based on a SQL table.

A typed view or subview must follow some specific rules:

- The *SELECT* statement of the view cannot reference *DBPARTITIONNUM* or *HASHEDVALUE* functions, functions with external actions, or any non-deterministic function.
- The view can contain a single *SELECT* statement. More *SELECT* statements can be added by using a *UNION ALL* clause between each one. (DB2 refers to a query with multiple *SELECT* statements as having multiple *branches*.)
- Branches should not cross hierarchies.
- Branches of a typed view should not contain *GROUP BY* or *HAVING* clauses.
- The root view should contain the *UNCHECKED* clause if it contains branches.

The following is an example including types, a typed table, and a view hierarchy, built around **people_type** and **emp_type**:

```
CREATE TYPE people_type
AS (name VARCHAR(20), address VARCHAR(100))
MODE DB2SQL
CAST (SOURCE AS REF) WITH people_func;

CREATE TYPE emp_type
UNDER people_type
AS (title VARCHAR(20), salary DECIMAL(8,2))
MODE DB2SQL
CAST (SOURCE AS REF) WITH emp_func;

CREATE TABLE people
OF people_type
(REF IS people_oid USER GENERATED);

INSERT INTO people (people_oid, name, address)
```



```
VALUES (people_func('0'), 'Sammy', '123 Lane');

CREATE TABLE emp
  OF emp_type
  UNDER people INHERIT SELECT PRIVILEGES;

INSERT INTO emp (people_oid, name, address, title, salary)
  VALUES (emp_func('1'), 'Josh', '123 Street', 'Director', 1000);

CREATE VIEW people_view
  OF people_type
  MODE DB2SQL
  (REF IS people_view_oid USER GENERATED)
  AS SELECT people_func (VARCHAR(people_oid)), name, address
     FROM ONLY (people);

CREATE VIEW emp_view
  OF emp_type
  MODE DB2SQL
  UNDER people_view INHERIT SELECT PRIVILEGES
  AS SELECT emp_func (VARCHAR(people_oid)), name,
     address, title, salary
     FROM emp;

SELECT *
  FROM people_view;

PEOPLE_VIEW_OID NAME ADDRESS
-----
x'30' Sammy 123 Lane
x'31' Josh 123 Street

SELECT *
  FROM emp_view;

PEOPLE_VIEW_OID NAME ADDRESS TITLE SALARY
-----
x'31' Josh 123 Street Director 1000.00
```

The *WITH CHECK OPTION* clause can be tricky because you are, in effect, telling the views to combine *WHERE* clauses either from upward in the dependency chain (via *CASCADE*) or downward in the dependency chain (via *LOCAL*). Consider the following:

```
V1 sourced from table T
V2 sourced from V1 WITH check_opn CHECK OPTION
V3 sourced from V2
V4 sourced from V3 WITH check_opn CHECK OPTION
V5 sourced from V4
```

The following table shows the search conditions against which inserted or updated rows are checked.



Note that if you create a view (or any other object, for that matter) with a schema name that does not already exist in the database, DB2 will implicitly create a schema of that name.

MySQL

Not supported.

Oracle

Oracle supports extensions to the ANSI standard to create object-oriented views, XMLType views, and views that support *LOB* and object types:

```
CREATE [OR REPLACE] [ [NO] FORCE] VIEW view_name
  {[(column [,...]) [constraint_clause] |
  [OF type_name {UNDER parent_view |
  WITH OBJECT IDENTIFIER {DEFAULT | (attribute [,...]) }
  [(constraint_clause)] ] |
  [OF XMLTYPE [ [XMLSCHEMA xml_schema_url] ELEMENT
  {element | xml_schema_url # element} ]
  WITH OBJECT IDENTIFIER {DEFAULT | (attribute [,...]) }
AS
(select statement)
[WITH [READ ONLY | CHECK OPTION [CONSTRAINT constraint_name] ] ]
```

Oracle's implementation of the *ALTER VIEW* statement supports added capabilities, such as adding, dropping, or modifying constraints associated with the view. In addition, the *ALTER VIEW* statement will explicitly recompile a view that is invalid, enabling you to locate recompile errors before runtime. This recompiling feature enables you to determine if a change in a base table negatively impacts any dependent views:

```
ALTER VIEW view_name
  {ADD constraint_clause |
  MODIFY CONSTRAINT constraint_clause [NO]RELY] |
  DROP {PRIMARY KEY | CONSTRAINT constraint_clause |
  UNIQUE (column [,...])} }
COMPILE
```

The parameters are:

OR REPLACE

Replaces any existing view of the same *view_name* with the new view.

[*NO*] *FORCE*

The *FORCE* clause creates the view regardless of whether the base tables exist or the user creating the view has privileges to the base tables, views, or functions defined in the view. The *FORCE* clause also creates the

view regardless of errors that occur during view creation. The *NO FORCE* clause creates the view only if the base tables and proper privileges are in place.

constraint_clause

Allows you to specify constraints (see *CREATE TABLE*) on views for *CREATE VIEW*. Using *ALTER VIEW*, this clause also allows you to affect a named, pre-existing constraint. You can define the constraint at the view level, similar to a table-level view, or at the column or attribute level. Note that although Oracle allows you to define constraints on a view, it doesn't yet enforce them. Oracle supports constraints on a view in *DISABLE* and *NOVALIDATE* modes.

OF *type_name*

Declares that the view is an object view of *type_name*. The columns of the view correspond directly to the attributes returned by *type_name*, where *type_name* is a previously declared type (see *CREATE TYPE*). You do not specify column names for object and XMLType views.

UNDER *parent_view*

Specifies a subview based on a pre-existing *parent_view*. The subview must be in the same schema as the parent view, the *type_name* must be an immediate subtype of the *parent_view*, and one subview is allowed.

WITH OBJECT IDENTIFIER {DEFAULT | (*attribute* [,...]) }

Defines the root object view as well as any *attribute* of the object type used to identify each row of the object view. The attributes usually correspond to the primary key columns of the base table and must uniquely identify each row of the view. This clause is incompatible with subviews and dereferenced or pinned *REF* keys. The *DEFAULT* keyword uses the implicit object identifier of the base object table or view.

OF XMLTYPE [[XMLSCHEMA *xml_schema_url*] ELEMENT { *element* | *xml_schema_url* # *element* } WITH OBJECT IDENTIFIER (*attribute* [,...])

Specifies that the view will return XMLType instances. Specifying the optional *xml_schema_url* as a pre-registered XMLSchema and element name further constrains the returned XML as an element in that XMLSchema. The *WITH OBJECT IDENTIFIER* clause specifies the identifier that uniquely identifies each row of the XMLType view. One or more attributes may use non-aggregate functions like **extractValue()** to obtain the identifiers from the resultant XMLType.

WITH READ ONLY

Ensures the view is used only to retrieve data and not to modify it.

WITH CHECK OPTION [CONSTRAINT *constraint_name*]

Forces the view to accept only inserted and updated data that can be returned by the view's *SELECT* statement. Alternately, you can specify a single *CHECK OPTION constraint_name* that exists on the base table that you want to enforce. If the constraint is not named, Oracle names the constraint *SYS_Cn*, where *n* is an integer.

ADD *constraint_clause*

Adds a new constraint to the view. Oracle supports constraints only in *DISABLE* and *NOVALIDATE* mode.

MODIFY CONSTRAINT *constraint_clause* [NO]RELY

Changes the *RELY* or *NORELY* setting of an existing view constraint. (*RELY* and *NORELY* are explained under *CREATE TABLE*.)

DROP {PRIMARY KEY | CONSTRAINT *constraint_clause* | UNIQUE (*column* [,...]) }

Drops an existing constraint on a view.

COMPILE

Recompiles the view.

Any dblink in the view's *SELECT* statement must be declared using the *CREATE DATABASE LINK...CONNECT TO* statement. Views containing flashback queries will have their *AS OF* clause evaluated at each invocation of the view, not when the view is compiled.

In this example, we create a view that has an added constraint:

```
CREATE VIEW california_authors (last_name, first_name,  
    author_ID UNIQU RELY DISABLE NOVALIDATE,  
    CONSTRAINT id_pk PRIMARY KEY (au_id) RELY DISABLE NOVALIDATE)  
AS  
SELECT au_lname, au_fname, au_id  
FROM authors  
WHERE state = 'CA';
```

We might also wish to create an object view on an Oracle database and schema. This example creates the type and the object view:

```
CREATE TYPE inventory_type AS OBJECT  
( title_id NUM(6),  
    warehouse wrhs_typ,  
    qty NUM(8) );  
CREATE VIEW inventories OF inventory_type  
WITH OBJECT IDENTIFIER (title_id)  
AS  
SELECT i.title_id, wrhs_typ(w.wrhs_id, w.wrhs_name,  
    w.location_id), i.qty  
FROM inventories i  
JOIN warehouses w ON i.wrhs_id = w.wrhs_id;
```

You could recompile the **inventory_type** view like this:

```
ALTER VIEW inventory_type COMPILE;
```

An updateable view in Oracle cannot contain any of the following:

- The *DISTINCT* clause
- *UNION*, *INTERSECT*, or *MINUS*
- Joins that cause inserted or updated data to affect more than one table
- Aggregate or analytic functions
- *GROUP BY*, *ORDER BY*, *CONNECT BY*, or *START WITH* clauses
- Subqueries or collection expressions in the *SELECT* item list (Subqueries are acceptable in the *SELECT* statement's *WHERE* clause.)
- Update pseudocolumns or expressions

There are some restrictions on how subviews and materialized views can be defined in Oracle:

- The subview must use aliases for **ROWID**, **ROWNUM**, or **LEVEL** pseudocolumns.
- The subview cannot query **CURVAL** or **NEXTVAL** pseudocolumns.
- The subview cannot contain the *SAMPLE* clause.
- The subview evaluates all columns of a *SELECT * FROM...* statement at compile time. Thus, any new columns added to the base table will not be retrieved by the subview until the view is recompiled.

Note that older versions of Oracle supported partitioned views. This feature has been deprecated. You should use explicitly declared partitions instead.

PostgreSQL

PostgreSQL supports a no-frills subset of the ANSI standard:

```
CREATE VIEW view_name [ (column [...]) ]
```

```
AS select_statement
```

PostgreSQL's *CREATE VIEW* does not support some of the more complex options that other vendors do. However, it does allow views to be built on tables and other defined class objects. PostgreSQL views are built only upon other tables, not upon other views. PostgreSQL views are always read-only and cannot be used to perform data modifications on the underlying base tables.

SQL Server

SQL Server supports some extensions to the ANSI standard, but does not offer object views or subviews:

```
CREATE VIEW view_name [(column [...])] 
```

```
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA} [...]]
```

```
AS
```

```
select_statement
```

```
[WITH CHECK OPTION]
```

SQL Server's implementation of *ALTER VIEW* allows you to change an existing view without affecting the permissions or dependent objects of the view:

```
ALTER VIEW view_name [(column [...])] 
```

```
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA} [...]]
```

```
AS
```

```
select_statement
```

```
[WITH CHECK OPTION]
```

The parameters are as follows:

ENCRYPTION

Encrypts the text of the view in the **syscomments** table. This option is usually invoked by software vendors who want to protect their intellectual capital.

SCHEMABINDING

Binds the view to a specific schema, meaning that all objects in the view must be referenced by their full name (both owner and object name). The view and any tables referenced in the view *SELECT* clause must be fully qualified, as in **pubs.scott.employee**. Views created with *SCHEMABINDING* (and tables referenced by these views) must have the schema binding dropped (via *ALTER VIEW*) before they may be dropped or altered.

VIEW_METADATA

Specifies that SQL Server return metadata about the view (rather than the base table) to calls made from DBLIB or OLEDB APIs. Views created or altered with *VIEW_METADATA* enable their columns to be updated by *INSERT* and *UPDATE INSTEAD OF* triggers.

WITH CHECK OPTION

Forces the view to accept only inserted and updated data that can be returned by the view's *SELECT* statement.

The *SELECT* clause of a SQL Server view cannot:

- Have the *COMPUTE*, *COMPUTE BY*, *INTO*, or *ORDER BY* clauses (*ORDER BY* is allowed if you use *SELECT TOP*)
- Reference a temporary table
- Reference a table variable
- Reference more than 1,024 columns, including those referenced by subqueries

Here, we define a SQL Server view with both *ENCRYPTION* and *CHECK OPTION*:

```
CREATE VIEW california_authors (last_name, first_name, author_id)
```

```
WITH ENCRYPTION
```

```
AS
```

```
    SELECT au_lname, au_fname, au_id
```

```
    FROM authors
```

```
    WHERE state = 'CA'
```

```
WITH CHECK OPTION
```

```
GO
```

SQL Server allows multiple *SELECT* statements in a view, as long as they are linked with *UNION* or *UNION ALL* clauses. SQL Server also allows functions and hints in the *SELECT* statement of a view. A SQL Server view is updateable if all of the items in the following list are true.

- The *SELECT* statement has no aggregate functions
- The *SELECT* statement does not contain *TOP*, *GROUP BY*, *DISTINCT*, or *UNION*
- The *SELECT* statement has no derived columns (see *SUBQUERY*)
- The *FROM* clause of the *SELECT* statement references at least one table

SQL Server allows indexes to be created on views (see *CREATE INDEX*). By creating a unique, clustered index on a view, you cause SQL Server to store a physical copy of the view on the database. Changes to the base table are automatically updated in the indexed view. Indexed views consume extra disk space, but provide a boost in performance. Indexed views must be built using the *SCHEMABINDING* clause.

SQL Server also allows the creation of local partitioned views and distributed partitioned views. A *local partitioned view* is a partitioned view where all views are present on the same SQL Server. A *distributed partitioned view* is a partitioned view where one or more views are located on remote servers.

Partitioned views must very clearly derive their data from different sources, with each distinct data source joined with a *UNION ALL* statement to the next. Furthermore, all columns of the partitioned views should be selected and identical.

(The idea is that you have split the data out logically by means of a frontend application. SQL Server then recombines the data through the partitioned view.) This example shows how the data in the view comes from three separate SQL Servers:

```
CREATE VIEW customers
AS
--Select from a local table on server 'New_York'
SELECT *
FROM sales_archive.dbo.customers_A
UNION ALL
SELECT *
FROM houston.sales_archive.dbo.customers_K
UNION ALL
SELECT *
FROM los_angeles.sales_archive.dbo.customers_S
```

Note that each remote server (*New_York*, *houston*, and *los_angeles*) has to be defined as a remote server on all of the SQL Servers using the distributed partitioned view. Partitioned views can greatly boost performance because they can split I/O and user loads across many machines. But they are also difficult to plan, create, and maintain. Be sure to read the vendor documentation for complete details about all the permutations available with partitioned views.

When altering an existing view, SQL Server acquires and holds an exclusive schema lock on the view until the alteration is finished. *ALTER VIEW* drops any indexes that might be associated with a view. You must manually recreate them using *CREATE INDEX*.

See Also

- CREATE/ALTER TABLE
- DROP
- SELECT
- SUBQUERY

DECLARE CURSOR Command

The *DECLARE* command is one of four commands used in cursor processing, along with *FETCH*, *OPEN*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than in a complete set. The *DECLARE CURSOR* command specifies the exact records to be retrieved and manipulated one row at a time from a specific table or view.

In other words, cursors are especially important for relational databases because databases are set-based, while most client-centric programming languages are row-based. This is important for two reasons. First, cursors allow programmers to program using methodologies supported by their favorite row-based programming language. Second, cursors run counter to the default behavior of some relational database platforms, which operate on a set of records and, on those specific platforms, may be noticeably slower than standard set-based operations.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

DECLARE *cursor_name* [{SENSITIVE | INSENSITIVE | ASENSITIVE}]

[[NO] SCROLL] CURSOR [{WITH | WITHOUT} HOLD]

[{WITH | WITHOUT} RETURN]

FOR *select_statement*

[FOR {READ ONLY | UPDATE [OF *column* [,...]]}]

Keywords

DECLARE *cursor_name*

Gives the cursor a unique name in the context in which it is defined — for example, in the database or schema where it is created. The name must be unique, meaning that no other cursors may share the same name.

SENSITIVE | *INSENSITIVE* | *ASENSITIVE*

Defines the manner in which the cursor interacts with the source table and the result set retrieved by the cursor query, where:

SENSITIVE

Tells the database to operate directly against the result set, such that the cursor will see changes to its result set as it moves through the records.

INSENSITIVE

Tells the database to create a temporary but separate copy of the result set, so that all changes made against the result set by other operations are invisible to the cursor.

ASENSITIVE

Allows the database implementation to determine whether to make a copy of the result set. *ASENSITIVE* is the SQL2003 default.

[NO] SCROLL

Tells the database not to enforce processing one row at a time, using *FETCH NEXT*, but that all forms of the *FETCH* clause are allowed against the result set. *NO SCROLL* enforces processing one row at a time.

{WITH | WITHOUT} HOLD

Tells the cursor to remain open when a transaction encounters the *COMMIT* statement. Conversely, *WITHOUT HOLD* tells the cursor to close when the transaction encounters a *COMMIT* statement. (A cursor that uses *WITH HOLD* can only be and is always closed with a *ROLLBACK* statement or a *CLOSE CURSOR* statement.)

{WITH | WITHOUT} RETURN

Used only in a stored procedure, the *WITH RETURN* clause tells the database to return the result set, if it is still open, when the stored procedure terminates. With the *WITHOUT RETURN* clause, which is the default behavior, all open cursors are implicitly closed when the stored procedure terminates.

FOR *select_statement*

Defines the underlying *SELECT* statement that determines the rows in the result set of the cursor. As with a regular *SELECT* statement, the results of the query may be sorted according to an *ORDER BY* clause.


```
FOR {READ ONLY | UPDATE [OF column [,...]] }
```

Specifies that the cursor is not updateable in any way, using *FOR READ ONLY*. This is the default when the cursor is defined with *SCROLL* or *INSENSITIVE* properties or the query contains an *ORDER BY* clause or is against a non-updateable table. Alternately, you can specify *FOR UPDATE OF column1, column2 [,...]*, defining the columns where you want to execute *UPDATE* statements. You can leave off a column list on the *FOR UPDATE OF* to include all columns in the cursor.

Rules at a Glance

The *DECLARE CURSOR* command enables the retrieval and manipulation of records from a table one row at a time. This provides row-by-row processing, rather than the traditional set processing offered by SQL.

At the highest level, these steps are followed when working with a cursor:

- Create a cursor using *DECLARE*.
- Open the cursor using *OPEN*.
- Operate against the cursor using *FETCH*.
- Dismiss the cursor using *CLOSE*.

The *DECLARE CURSOR* command works by specifying a *SELECT* statement. Each row returned by the *SELECT* statement may be individually retrieved and manipulated. In this Oracle example, the cursor is declared in the declaration block along with some other variables. The cursor is then opened, fetched against, and then closed in the *BEGIN...END* block that follows:

DECLARE

```
CURSOR title_price_cursor IS
```

```
SELECT title, price
```

```
FROM titles
```

```
WHERE price IS NOT NULL;
```

```
title_price_val title_price_cursor%ROWTYPE;
```

```
new_price NUMBER(10,2);
```

```
BEGIN
```

```
OPEN title_price_cursor;
```

```
FETCH title_price_cursor INTO title_price_val;
```

```
new_price := "title_price_val.price" * 1.25
```

```
INSERT INTO new_title_price
```

```
VALUES (title_price_val.title, new_price)
```

```
CLOSE title_price_cursor;
```

```
END;
```

Because this example uses PL/SQL, much of the code is beyond the scope of this book. However, the *DECLARE* block clearly shows that the cursor is declared. In the PL/SQL execution block, the cursor is initialized with the *OPEN* command, values are retrieved with the *FETCH* command, and the cursor finally is terminated with the *CLOSE* command.

The *SELECT* statement is the heart of your cursor, so it is a good idea to test it thoroughly before embedding it in the *DECLARE CURSOR* statement. The *SELECT* statement may operate against a base table or a view. For that matter, read-only cursors may operate against non-updateable views. The *SELECT* statement can have subclauses like *ORDER BY*, *GROUP BY*, and *HAVING* if it is not updating the base table. If the cursor is *FOR UPDATE*, then it is a good idea to exclude these clauses from the *SELECT* statement.

Local cursors are sometimes used as output parameters of stored procedures. Thus, a stored procedure could define and populate a cursor and pass that cursor to a calling batch or stored procedure.

In this simple DB2 example, we'll declare a cursor that looks at department numbers, names, and manager numbers for all departments in the **admin_group 'X01'**:

```
EXEC SQL
DECLARE dept_cursor CURSOR
FOR SELECT dept_nbr, dept_name, mgr_nbr
      FROM department
      WHERE admin_group = 'X01'
      ORDER BY dept_name ASC, dept_nbr DESC, mgr_nbr DESC;
```

In this next example from Microsoft SQL Server, a cursor from the **publishers** table is declared and opened. The cursor takes the first record from **publishers** that matches the *SELECT* statement and inserts it into another table; it then moves to the next record and the next, until all records are processed. Finally, the cursor is closed and deallocated (*DEALLOCATE* is used only with Microsoft SQL Server):

```
DECLARE @publisher_name VARCHAR(20)
DECLARE pub_cursor CURSOR
FOR SELECT pub_name FROM publishers
      WHERE country <> 'USA'
OPEN pub_cursor
FETCH NEXT FROM pub_cursor INTO @publisher_name
WHILE @@FETCH_STATUS = 0
BEGIN
      INSERT INTO foreign_publishers VALUES(@publisher_name)
END
CLOSE pub_cursor
DEALLOCATE pub_cursor
```

In this example, you see how the cursor moves through a set of records. (The example was intended only to demonstrate this concept, since there is actually a better way to accomplish the same task, namely an *INSERT...SELECT* statement.)

Programming Tips and Gotchas

Most platforms do not support dynamically executed cursors. Rather, cursors are embedded within an application program, stored procedure, user-defined function, etc. In other words, the various statements for creating and using a cursor are usually used only in the context of a stored procedure or other database-programming object, not in a straight SQL script.

For ease of programming and migration, don't use the *SCROLL* clause or the various sensitivity keywords. Some platforms support only forward-scrolling cursors, so you can avoid headaches later by just sticking to the simplest form of cursor.

Cursors behave differently on the various database platforms when crossing a transaction boundary — for example, when a set of transactions is rolled back in the midst of a cursor. Be sure to get familiar with exactly how each database platform behaves in these circumstances.

MySQL does not support server-side cursors in the ANSI SQL style, but does support extensive C-programming extensions that provide the same functionality.

DB2

DB2 supports the basic *DECLARE CURSOR* statement along with some variations on the *WITH HOLD* and *WITH RETURN* clauses, as in the following example:

```
DECLARE cursor_name CURSOR  
  
[WITH HOLD] [WITH RETURN [{TO CALLER | TO CLIENT}]]  
  
FOR select_statement
```

where:

WITH RETURN [{TO CALLER | TO CLIENT}]

Used only in a stored procedure, the *WITH RETURN* clause tells the database to return the result set, if it is still open, when the stored procedure terminates. (DB2 will accept *WITH RETURN* on cursors outside of a stored procedure, but it has no effect.) It has these options:

TO CALLER

Returns the result set to the caller, such as another stored procedure or process on the database server. It will also return the result set to a client, if that is what called the stored procedure. This is the default behavior for cursors embedded in DB2 external stored procedures.

TO CLIENT

Returns the result set to the client application, bypassing any intermediate processes such as nested stored procedures. This clause cannot be used to return result sets to functions or methods.

FOR *select_statement*

Defines the underlying *SELECT* statement that determines the rows in the result set of the cursor. The *SELECT* statement cannot include parameters, but can include host variables, as long as they have been declared before the cursor. You may also use a prepared *SELECT* statement as the *select_statement*, using the DB2 *PREPARE* command.

DB2 provides syntax to explicitly declare a cursor as read-only or updateable as part of the *SELECT* statement rather than as part of the *DECLARE CURSOR* statement.

When not specified, the updateability of a cursor is implicitly determined according to a variety of rules that improve locking. First of all, a cursor is assumed to be *read-only* if it is not deleteable. A DB2 cursor is *deleteable* when it has the *FOR UPDATE* clause plus:

- The *FROM* clause of the *SELECT* statement references only one base table or deleteable view.
- The *SELECT* statement does not contain a *VALUES* clause, a *DISTINCT* clause, a *GROUP BY* clause, a *HAVING* clause, an *ORDER BY* clause, a *FOR READ ONLY* clause, or *UNION*, *EXCEPT*, or *INTERSECT* clauses.
- The *SELECT* statement is free of column functions in the select item list.

The columns of a DB2 cursor are *updateable* when it meets the requirements to be deleteable plus:

- The columns are resolved to columns in the base table or view.
- The *LANGLEVEL* bind option is *MIA*, *SQL92E*, or the columns are referenced in the *FOR UPDATE* clause of the *select_statement*.



A DB2 cursor may also be updateable if it is statically defined or has the SQL-standard bind option (*MIA*) applied to *LANGLEVEL*. The *LANGLEVEL* setting *MIA* indicates that the cursor runs under SQL92 rules. The setting *SQL92E* indicates that the cursor runs under the SQL92 entry-level rules. And the setting *SAA1* indicates that the cursor runs under the DB2 common (old) rules. *LANGLEVEL* is assigned using the DB2 *PREP* command.

DB2 also allows cursors to be *ambiguous*. An ambiguous cursor is considered updateable unless the *BLOCKING* bind option is *ALL*, whence it is considered read-only. A cursor is ambiguous when all these conditions are true:

- The *select_statement* doesn't include a *FOR READ ONLY* or *FOR UPDATE* clause.
- The language level setting (*LANGLEVEL*) is set to the value *SAA1*.
- The *select_statement* is dynamically prepared.
- The cursor is otherwise deletable.

There are a couple things to be careful of with DB2 cursors:

- System function values, such as those returned by *CURRENT DATE*, *CURRENT TIME*, or *CURRENT TIMESTAMP*, are determined when a cursor is opened and not when the *FETCH* statement is executed. Therefore, any of these values would be identical for every record in the result set, since the values are determined once when the cursor is opened.
- If a *LANGUAGE* other than SQL (such as Java) is used for the stored procedure, the default behavior is *WITH RETURN TO CALLER*.

MySQL

Not supported.

Oracle

Oracle has a rather interesting implementation of cursors. In reality, all Oracle data modification statements (*INSERT*, *UPDATE*, *DELETE*, and *SELECT*) implicitly open a cursor. For example, a C program accessing Oracle would not need to issue a *DECLARE CURSOR* statement to retrieve data on a record-by-record basis because that is the implicit behavior of Oracle. Because of this behavior, *DECLARE CURSOR* is something you use only in PL/SQL constructs, like stored procedures, and not something you would use in a script that is purely SQL.



Since cursors can only be used in stored procedures and user-defined functions on Oracle, they are documented in the Oracle PL/SQL reference material, not in the SQL reference material.

Oracle utilizes a variation of the *DECLARE CURSOR* statement that supports parameterized inputs, as in the following syntax:

```
DECLARE CURSOR cursor_name [ ( parameter datatype [,...] ) ]
```

```
IS select_statement
```

```
[FOR UPDATE [OF column_name [,...]]]
```

where:

```
parameter datatype [,...] IS select_statement
```

Defines the *select_statement* used to retrieve the cursor result set. It serves the same purpose as the ANSI SQL2003 *FOR select_statement* clause.

```
FOR UPDATE [OF column_name]
```

Defines the cursor or specific columns of the cursor as updateable. Otherwise, the cursor is assumed to be read-only.

In Oracle, variables are not allowed in the *WHERE* clause of the *SELECT* statement unless they are first declared as variables. The parameters are not assigned at the *DECLARE*; instead, they are assigned values at the *OPEN* command. This is important since any system function will return an identical value for all rows in the cursor result set.

PostgreSQL

PostgreSQL does not support the *WITH* clauses. It does allow you to return result sets in *BINARY* rather than text format. Although the compiler will not cause errors with many of the ANSI keywords, PostgreSQL's implementation of *DECLARE CURSOR* is more limited than it might first appear:

```
DECLARE cursor_name [BINARY] [INSENSITIVE] [SCROLL] CURSOR
```

```
FOR select_statement
```

```
[FOR {READ ONLY | UPDATE [OF column_name [...]]}]
```

where:

BINARY

Forces the cursor to retrieve binary-formatted data rather than text-formatted data.

INSENSITIVE

Indicates that data retrieved by the cursor is unaffected by updates from other processes or cursors. However, this is PostgreSQL's default behavior, so adding or leaving this keyword off has no effect.

SCROLL

Allows multiple rows to be retrieved by a single *FETCH* operation. However, this is PostgreSQL's default behavior, so adding or leaving off this keyword has no effect.

```
FOR {READ-ONLY | UPDATE}
```

Indicates that a cursor is opened using read-only mode or updateable mode, respectively. However, PostgreSQL only supports read-only cursors. That means that the *FOR READ-ONLY* clause has no effect, while the *FOR UPDATE* (*column_name*) clause produces an informational error message.

PostgreSQL closes any existing cursor when a newly declared cursor is created with the same name. Binary cursors tend to be faster because PostgreSQL stores data as binary on the backend. However, user applications are only text aware. Therefore, you would have to build in binary handling for any *BINARY* cursors.

PostgreSQL allows cursors only within a transaction. You should enclose a cursor within a *BEGIN* and *COMMIT* or *ROLLBACK* transaction block.

PostgreSQL does not support an explicit *OPEN* cursor statement. Cursors are considered open when they are declared. So to declare and open a cursor in PostgreSQL, you could use code like this:

```
DECLARE pub_cursor CURSOR
```

```
FOR SELECT pub_name FROM publishers
```

```
WHERE country <> 'USA';
```

SQL Server

SQL Server supports the ANSI standard, as well as a good many extensions that provide flexibility in how a cursor scrolls through a result set and manipulates data, as in the following syntax:

```
DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR
```

[LOCAL | GLOBAL] [FORWARD_ONLY | SCROLL]

[STATIC | KEYSSET | DYNAMIC | FAST_FORWARD]

[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]

[TYPE_WARNING]

FOR *select_statement*

[FOR {READ ONLY | UPDATE [OF *column_name* [,...]]}]]

The parameters are as follows:

LOCAL | *GLOBAL*

Scopes the cursor for either the local Transact-SQL batch or makes the cursor available to all Transact-SQL batches issued by the current session via *OPEN* and *FETCH* statements, respectively. A global cursor name can be referenced by any stored procedure, function, or Transact-SQL batch executed in the current session. A global cursor is implicitly deallocated at disconnect, but a local cursor must be manually deallocated. The *LOCAL* and *GLOBAL* keywords are optional. The default behavior, if neither is specified, is defined by the *default to local* database property.

INSENSITIVE | *FORWARD_ONLY* | *SCROLL*

Determines how the cursor will move through the result set, with three options:

INSENSITIVE

Creates the result set as a table in **TEMPDB**. Changes to the base table are not reflected in the cursor result set. Not compatible with SQL Server extensions to the *DECLARE CURSOR* command like *LOCAL*, *GLOBAL*, *STATIC*, *KEYSET*, *DYNAMIC*, *FAST_FORWARD*, etc. Can only be used in a SQL92-style *DECLARE CURSOR* statement, such as *DECLARE sample CURSOR INSENSITIVE FOR select_statement FOR UPDATE*.

FORWARD_ONLY

Indicates that the cursor must scroll from the first to the last record in the result set and that *FETCH NEXT* is the only supported form of the *FETCH* statement. The cursor is assumed to be *DYNAMIC* unless *STATIC* or *KEYSET* are used. *FAST_FORWARD* is mutually exclusive of *FORWARD_ONLY*.

SCROLL

Enables all *FETCH* options (*ABSOLUTE*, *FIRST*, *LAST*, *NEXT*, *PRIOR*, and *RELATIVE*). Otherwise, only *FETCH NEXT* is available. If *DYNAMIC*, *STATIC*, or *KEYSET* are used, the cursor defaults to *SCROLL* behavior. *FAST_FORWARD* is mutually exclusive of *SCROLL*.

STATIC | *KEYSET* | *DYNAMIC* | *FAST_FORWARD*

Determines how records are manipulated in the result set. These settings are incompatible with the *FOR READ ONLY* and *FOR UPDATE* clauses. There are four options:

STATIC

Makes a temporary copy of the result set data and stores it in *tempdb*. Modifications made to the source table or view do not show up when processing the cursor. *STATIC* cursors cannot modify data in the source table or view.

KEYSET

Makes a temporary copy of the result set with membership and fixed row order (also known as a keyset) in *tempdb*. The keyset correlates the data in the cursor result set to the base table or view such

that the cursor can see changes made to the underlying data. Rows that have been deleted or updated show an @@FETCH_STATUS of -2 (unless the update was done using *UPDATE...WHERE CURRENT OF*, in which case they are fully visible), while rows inserted by other users are not visible at all.

DYNAMIC

Determines the records in the cursor result set as each *FETCH* operation is executed. Thus, *DYNAMIC* cursors see all changes made to the base table or view, even those by other users. Because the result set may be in constant flux, *DYNAMIC* cursors do not support *FETCH ABSOLUTE*.

FAST_FORWARD

Creates a *FORWARD_ONLY*, *READ_ONLY* cursor that quickly reads through the entire cursor result set at one time.

READ_ONLY | *SCROLL_LOCKS* | *OPTIMISTIC*

Determines concurrency and positional update behavior for the cursor. These settings are incompatible with *FOR READ ONLY* and *FOR UPDATE*. The allowable parameters include:

READ_ONLY

Prevents updates to the cursor and disallows the cursor from being referenced in *UPDATE* or *DELETE* statements that contain *WHERE CURRENT OF*.

SCROLL_LOCKS

Ensures that positional updates and deletes succeed by locking the cursor result set rows as they are read into the cursor. The locks are held until the cursor is closed and deallocated. *SCROLL_LOCKS* is mutually exclusive of *FAST_FORWARD*.

OPTIMISTIC

Ensures that positional updates and deletes succeed unless the row being updated or deleted in the cursor result set has changed since it was read into the cursor. SQL Server accomplishes this by comparing a timestamp, or a checksum value when no timestamp exists, on the columns. It does not lock the rows in the cursor result set. *OPTIMISTIC* is mutually exclusive of *FAST_FORWARD*.

TYPE_WARNING

Warns the user when the cursor is implicitly converted from one type to another — for example from *SCROLL* to *FORWARD_ONLY*.

FOR {READ ONLY |

Identifies the cursor as read only using the ANSI standard syntax. This keyword is not compatible with the other type identifiers shown previously. It should only be used with *INSENSITIVE*, *FORWARD_ONLY*, and *SCROLL*.

FOR UPDATE [OF *column_name* [,...]]}

Allows updates to columns in the cursor using *UPDATE* and *DELETE* statements with the *WHERE CURRENT OF* clause. If *FOR UPDATE* is used without a column list, then all columns in the cursor are updateable. Otherwise, only those columns listed are updateable.



Microsoft SQL Server allows two basic forms of syntax for the *DECLARE CURSOR* statement. *The syntax forms are not compatible!* The basic forms of syntax are SQL92 compatible and Transact-SQL extensions. You *cannot* mix keywords of the two forms together.

The SQL-92 compatibility syntax for *DECLARE CURSOR* is:

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [,...] ] } ]
```

While the Transact-SQL extended syntax for *DECLARE CURSOR* is:

```
DECLARE cursor_name CURSOR
[ LOCAL | GLOBAL ] [ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [,...] ] ]
```

The SQL92 compatibility syntax is supported to enable your code to be more transportable. The Transact-SQL extensions syntax enables you to define cursors using the same cursor types that are available in popular database APIs like ODBC, OLE-DB, and ADO, but may only be used in SQL Server stored procedures, user-defined functions, triggers, and ad hoc queries.

If you do define a Transact-SQL extension cursor but do not define concurrency behavior using the *OPTIMISTIC*, *READ_ONLY*, or *SCROLL_LOCKS* keywords, then:

- The cursor defaults to *READ-ONLY* if the cursor is defined as *FAST_FORWARD*, *STATIC*, or the user has insufficient privileges to update the base table or view.
- The cursor defaults to *OPTIMISTIC* if the cursor has been defined as *DYNAMIC* or *KEYSET*.

Note that variables may be used in the *select_statement* of a SQL Server cursor, but the variables are evaluated when the cursor is declared. Thus, a cursor containing the column based on the system function *GETDATE*() will have the same date and time for every record in the cursor result set.

In the following SQL Server example, we use a *KEYSET* cursor to change blank spaces to dashes in the **phone** column of the **authors** table:

```
SET NOCOUNT ON

DECLARE author_name_cursor CURSOR LOCAL KEYSSET TYPE_WARNING
    FOR SELECT au_fname FROM pubs.dbo.authors

DECLARE @name varchar(40)

OPEN author_name_cursor

FETCH NEXT FROM author_name_cursor INTO @name

WHILE (@@fetch_status <> -1)

BEGIN
-- @@fetch_status checks for 'record not found' conditions and errors
    IF (@@fetch_status <> -2)
        BEGIN
            PRINT 'updating record for ' + @name
            UPDATE pubs.dbo.authors
            SET phone = replace(phone, ' ', '-')
            WHERE CURRENT OF author_name_cursor
```



```
END  
  
FETCH NEXT FROM author_name_cursor INTO @name  
  
END  
  
CLOSE author_name_cursor  
  
DEALLOCATE author_name_cursor  
  
GO
```

See Also

CLOSE CURSOR
FETCH
OPEN

DELETE Statement

The *DELETE* statement erases records from a specified table or tables. *DELETE* statements acting against tables are sometimes called *search deletes*. The *DELETE* statement may also be used in conjunction with a cursor. *DELETE* statements acting upon the rows of a cursor are sometimes called *positional deletes*.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DELETE FROM { table_name | ONLY (table_name) }  
  
[ { WHERE search_condition | WHERE CURRENT OF cursor_name } ]
```

Keywords

FROM *table_name*

Identifies the table, called *table_name*, from which rows will be deleted. The *table_name* assumes the current schema if one is not specified. You may alternately specify a single table view name. *FROM* is mandatory, except in the *DELETE...WHERE CURRENT OF* statement. When not using the *ONLY* clause, do not enclose the *table_name* in parentheses.

ONLY (*table_name*)

Restricts cascading of the deleted records to any subtables of the target table or view. *ONLY* effects only typed (object-oriented) tables and views. If used with a non-typed table or view, it is ignored and does not cause an error. If *ONLY* is used, you must enclose the *table_name* in parentheses.

WHERE *search_condition*

Defines search criteria for the *DELETE* statement using one or more *search_condition* clauses to ensure that only the target rows are deleted. Any legal *WHERE* clause is acceptable. Typically, these criteria are evaluated against each row of the table before the deletion occurs.

WHERE CURRENT OF *cursor_name*

Restricts the *DELETE* statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

The *DELETE* statement erases rows from a table or view. Space released by erased rows will be returned to the database where the table is located, though it may not happen immediately.

A simple *DELETE* statement that erases all records in a given table has no *WHERE* clause, as in the following:

```
DELETE FROM sales;
```

You can use any valid *WHERE* clause to filter records that you do not want to delete. All three of the following examples show valid *DELETE* statements, and since all are search deletes, they all include the *FROM* clause:

```
DELETE FROM sales
```

```
WHERE qty IS NULL;
```

```
DELETE FROM suppliers
```

```
WHERE supplierid = 17
```

```
OR companyname = 'Tokyo Traders';
```

```
DELETE FROM distributors
```

```
WHERE postalcode IN
```

```
(SELECT territorydescription FROM territories);
```

In some cases, you may wish to delete a specific row that is being processed by a declared and open cursor:

```
DELETE titles WHERE CURRENT OF title_cursor;
```

The above query assumes that you have declared and opened a cursor named **title_cursor**. Whichever row the cursor is on will be deleted when the command, shown above, is executed. Note that in the positional delete shown earlier, the *FROM* clause is not required.

Programming Tips and Gotchas

It is rare to issue a *DELETE* statement without a *WHERE* clause, because this results in deleting *all* rows from the affected table. You should first issue a *SELECT* statement with the same *WHERE* clause you intend to use on the *DELETE* statement. That way you can be sure exactly which records will be deleted.

If it becomes necessary to remove all the rows in a table, you should consider using the non-ANSI *TRUNCATE TABLE* statement. In those databases that support the command, *TRUNCATE TABLE* is usually a faster method to physically remove all rows. *TRUNCATE TABLE* is faster than *DELETE* because the deletion of individual records is not logged. On some platforms, this makes rollback of a *TRUNCATE* statement impossible. The reduction of logging overhead saves considerable time when erasing a large number of records. On some database platforms, all foreign keys on the table must be dropped before issuing a *TRUNCATE* statement.

DB2

DB2 supports the ANSI standard, with the addition of transaction isolation control and the ability to assign an alias to the table from which records will be deleted, as in the following:

```
DELETE FROM { table_name | ONLY (table_name) } [AS alias]
```

[WHERE {*search_conditions* | CURRENT OF *cursor_name*}]

[WITH { RR | RS | CS | UR }]

The parameters are:

AS *alias*

Allows the target *table_name* to use an alias. This is especially useful for correlated subqueries. (Note that DB2 has a special object called an *alias* that may be created with the DB2 *CREATE ALIAS* statement. In this case, the term "alias" represents what the ANSI SQL standard calls a *correlation name*, something akin to a simple nickname for the table or view.)

WITH {RR | RS | CS | UR}

Controls transaction isolation level for the update statement. *RR* is repeatable read, *RS* is read stability, *CS* is cursor stability, and *UR* is uncommitted read. See the [SET TRANSACTION Statement](#) section for full details on these transaction isolation levels.

On DB2, the target may be a table, updateable view, or nickname. As the ANSI standard dictates, rows will be deleted from any subtables of the target table unless you use the *ONLY* keyword. If *ONLY* is used, you must enclose the *table_name* in parentheses.

If you use *ONLY* in combination with *WHERE CURRENT OF*, then the *SELECT* statement that defined the cursor must use the *FROM ONLY* clause as well. For example:

```
DECLARE CURSOR emp_cursor FOR
```

```
  SELECT name, address
```

```
  FROM ONLY (employee);
```

```
DELETE FROM ONLY (employee)
```

```
  WHERE CURRENT OF emp_cursor;
```

The previous code assumes that **employee** is the base table, with subtables below it. The cursor will only select rows that are contained in **employee** and not the other types contained in the subtables (e.g., **parttime**, **salaried**, **student**). Similarly, deletes will only take place at this level of the hierarchy as well.

MySQL

MySQL allows a number of extensions over the ANSI standard, but does not support the *WHERE CURRENT OF* clause, as in the following:

```
DELETE [LOW_PRIORITY] [QUICK] [table_name[*] [...] ]
```

```
{FROM table_name[*] [...] | [USING table_name[*] [...] ] }
```

```
[WHERE search_condition]
```

```
[ORDER BY clause]
```

```
[LIMIT nbr_of_rows]
```

The parameters are:

LOW PRIORITY

Delays the execution of *DELETE* until no other clients are reading from the table.

QUICK

Prevents the storage engine from merging index leaves during the delete operation.

DELETE *table_name* [,...]

Enables you to delete from more than one table at a time. Tables listed before the *FROM* clause, assuming one or more tables appear in the *FROM* clause, will be the target of the delete operation. If more than one table appears before the *FROM* clause, then all matching records in all tables will be deleted.

FROM *table_name*[*]

Specifies the table or tables where records will be deleted. (The *.** clause is an option to improve compatibility with MS-Access.) If tables are listed before the *FROM* clause, then the table or tables in the *FROM* clause are assumed to be used to support a join or lookup operation.

USING *table_name*[*] [,...]

Substitute the table or tables before the *FROM* clause with those after the *FROM* clause.

ORDER BY *clause*

Specifies the order that rows will be deleted in. This is useful only in conjunction with *LIMIT*.

LIMIT *nbr_of_row*

MySQL also can place an arbitrary cap on the number of records deleted before control is passed back to the client using the *LIMIT nbr_of_rows* clause.

MySQL allows deletion from more than one table at a time. For example, the following two *DELETE* statements are functionally equivalent:

```
DELETE orders FROM customers, orders
```

```
WHERE customers.customerid = orders.customerid
```

```
AND orders.orderdate BETWEEN '19950101' AND '19951231'
```

```
DELETE FROM orders USING customers, orders
```

```
WHERE customers.customerid = orders.customerid
```

```
AND orders.orderdate BETWEEN '19950101' AND '19951231'
```

In the examples above, we delete all *orders* made by *customers* during the year 1995. Note that you cannot use *ORDER BY* or *LIMIT* clauses in a multitable delete like those shown above.

You can also delete records in orderly batches using the *ORDER BY* clause in conjunction with the *LIMIT* clause:

```
DELETE FROM sales
```

```
WHERE customerid = 'TORTU'
```

```
ORDER BY customerid
```

```
LIMIT 5
```

MySQL exhibits several behaviors that speed up the delete operation. For example, MySQL returns the number of records deleted when it completes the operation. However, MySQL will return zero as the number of deleted records when you delete all records from a table because it is faster for MySQL to return zero than to count the rows deleted. In *AUTOCOMMIT* mode, MySQL will even substitute a *TRUNCATE* statement for a *DELETE* statement without a *WHERE* clause because *TRUNCATE* is faster.

Note that the speed of a MySQL delete operation is directly related to the number of indexes on the table and the available index cache. You can speed up delete operations by executing the command against tables with few or no indexes or by increasing the size of the index cache.

Oracle

Oracle allows you to delete rows from tables, views, materialized views, nested subqueries, and partitioned views and tables, as follows:

DELETE [FROM]

```
{table_name | ONLY (table_name)} [alias]
[({PARTITION (partition_name) |
SUBPARTITION (subpartition_name)} |
(subquery [WITH {READ ONLY |
CHECK OPTION [CONSTRAINT constraint_name]}] ) |
TABLE (collection_expression) [ (+) ] }
[WHERE search_condition]
[RETURNING expression [,...] INTO variable [,...]]
```

The parameters are:

table_name [*alias*]

Specifies the table, view, materialized view, or partitioned table or view where records will be deleted. You may optionally prepend a schema identifier to the *table_name* or append a database link. Otherwise, Oracle will assume the user's default schema and the local database server. You may optionally apply an *alias* to the *table_name*. The alias is required if the target table references an object type attribute or method.

PARTITION *partition_name*

Applies the delete operation to the named partition, rather than to the entire table. You are not required to name a partition when deleting from a partitioned table, but it can, in many cases, help reduce the complexity of the *WHERE* clause.

SUBPARTITION *subpartition_name*

Applies to a named subpartition, rather than to the entire table.

(*subquery* [WITH {READ ONLY | CHECK OPTION [CONSTRAINT *constraint_name*] }])

Specifies that the target for deletion is a nested subquery, not a table, view, or other database object. The parameters of this clause are:

subquery

Describes the *SELECT* statement that makes up the subquery. The subquery can be any standard subquery, though it may not contain an *ORDER BY* clause.

WITH READ ONLY

Specifies that the *subquery* cannot be updated.

WITH CHECK OPTION

Tells Oracle to abort any changes to the deleted table that would not appear in the result set of the subquery.

CONSTRAINT *constraint_name*

Tells Oracle to further restrict changes based upon a specific constraint identified by *constraint_name*.

TABLE (*collection_expression*) [(+)]

Informs Oracle that the *collection_expression* should be treated like a table though it may, in fact, be a subquery, a function, or some other collection constructor. In any case, the value returned by the *collection_expression* must be a nested table or *VARRAY*.

RETURNING *expression*

Retrieves the rows affected by the command, where *DELETE* normally only shows the number of rows deleted. The *RETURNING* clause can be used when the target is a table, a materialized view, or a view with a single base table. When used for single-row deletes, the *RETURNING* clause stores values from the row deleted by the statement, defined by *expression*, into PL/SQL variables and bind variables. When used for a multirow delete, the *RETURNING* clause stores the values of the deleted rows, defined by *expression*, into bind arrays.

INTO *variable*

Specifies the variables into which the values returned as a result of the *RETURNING* clause are stored. There must be a corresponding *variable* for every *expression* in the *RETURNING* clause.

When you execute a *DELETE* statement, Oracle releases space from the target table (or the base table of a target view) back to the table or index that owned the data.

When deleting from a view, the view cannot contain a set operator, the *DISTINCT* keyword, joins, an aggregate function, an analytic function, a subquery in the *SELECT* list, a collection expression in the *SELECT* list, a *GROUP BY* clause, an *ORDER BY* clause, a *CONNECT BY* clause, or a *START WITH* clause.

Here's an example where we delete records from a remote server:

```
DELETE FROM scott.sales@chicago;
```

In the following query, we delete from a derived table that is a collection expression:

```
DELETE TABLE(SELECT contactname FROM customers
               c WHERE c.customerid = 'BOTTM') s
WHERE s.region IS NULL OR s.country = 'MEXICO';
```

Here's an example where we delete from a partition:

```
DELETE FROM sales PARTITION (sales_q3_1997)
WHERE qty > 10000;
```

Finally, in the example below, we use the *RETURNING* clause to look at the values that are deleted:

```
DELETE FROM employee
WHERE job_id = 13
   AND hire_date + TO_YMINTERVAL('01-06') =<= SYSDATE;
RETURNING job_lvl
INTO :int01;
```

The previous example deletes records from the **employee** table and returns the **job_lvl** values into the predefined **:into1** variable.

PostgreSQL

PostgreSQL uses the *DELETE* command to remove rows and any defined subclasses from the table. It is otherwise identical to the ANSI standard, as follows:

```
DELETE [FROM] [ONLY] table_name  
[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

When deleting rows from only the table specified, use the optional *ONLY* clause. Otherwise, PostgreSQL will also delete records from any explicitly defined subtable.

To delete all records from the **titles** table:

```
DELETE titles
```

To delete all records in the **authors** table where the last name starts with "Mc":

```
DELETE FROM authors  
WHERE au_lname LIKE 'Mc%'
```

To delete all **titles** with an old ID number:

```
DELETE titles WHERE title_id >= 40
```

To delete all **titles** that have no sales:

```
DELETE titles WHERE ytd_sales IS NULL
```

To delete all records in one table based on the results of a subquery against another table (in this case, the records that have a match concerning "computers" in the **titles** table are erased from the **titleauthor** table):

```
DELETE FROM titleauthor  
WHERE title_id IN  
(SELECT title_id  
FROM titles  
WHERE title LIKE '%computers%')  
DISCONNECT
```

SQL Server

Microsoft SQL Server allows records to be deleted both from tables and from views that describe a single table. SQL Server also allows a second *FROM* clause to allow *JOIN* constructs, as in the following example:

```
DELETE [TOP number [PERCENT]] [FROM] table_name [[AS] alias]  
[FROM table_source [...]]  
[ [{INNER | CROSS | [{LEFT | RIGHT | FULL} OUTER}]  
JOIN joined_table ON condition][,...]  
[WHERE search_condition | WHERE CURRENT OF [GLOBAL] cursor_name ]  
[OPTION (query_hint[,...n])]
```

The syntax elements are as follows:

DELETE *table_name*

Allows the deletion of records from the named table or view filtered by the *WHERE* clause. You can delete records from a view providing the view is based on one table and contains no aggregate functions and no derived columns. If you leave off the server name, database name, or schema name when naming the table or view, SQL Server assumes the current context. An *OPENDATASOURCE* or *OPENQUERY* function, as described in the [SELECT Statement](#) section may be referenced instead of a table or view.

TOP *number* [PERCENT]

Specifies that only the *number* of rows are deleted by the statement. If *PERCENT* is specified, only the first *number* percent of the rows are retrieved. If the *number* is an expression, such as a variable, it must be enclosed in parentheses. The expression should be of *FLOAT* datatype with a range of 0 to 100 when using *PERCENT*. When not using *PERCENT*, the expression should be of *BIGINT* datatype.

FROM *table_source*

Names an additional *FROM* clause that correlates records from the table in the first *FROM* clause using a *JOIN* rather than forcing you to use a correlated subquery. One or more tables may be listed in the second *FROM* clause.

[{INNER | CROSS | [{LEFT | RIGHT | FULL} OUTER]}] JOIN *joined_table* ON *condition*[,...]

Specifies one or more *JOIN* clauses, in conjunction with the second *FROM* clause. You may use any of the join types that SQL Server supports. Refer to the [JOIN Subclause](#) and [SELECT Statement](#) sections for more information.

GLOBAL *cursor_name*

Specifies that the delete operation should occur on the current row of an open global cursor. This clause is otherwise the same as the standard for *WHERE CURRENT OF*.

OPTION (*hint* [...])

Replaces elements of the default query plan with your own. Because the optimizer usually picks the best query plan for any query, we strongly discourage you from placing optimizer hints into your queries.

The primary addition to *DELETE* that SQL Server offers over and above the ANSI standard is a second *FROM* clause. The second *FROM* allows the use of the *JOIN* statement and makes it quite easy to delete rows from the table in the first *FROM* by correlating rows of a table declared in the second *FROM*. For example, you can use a rather complex subquery to erase all the **sales** records of computer books with this command:

```
DELETE sales
WHERE title_id IN
    (SELECT title_id
     FROM titles
     WHERE type = 'computer')
```

But SQL Server allows a more elegant construction using a second *FROM* clause and a *JOIN* clause:

```
DELETE s
FROM sales AS s
INNER JOIN titles AS t ON s.title_id = t.title_id
    AND type = 'computer'
```

The following example deletes all rows with an **order_date** of 2003 or earlier in batches of 2,500:


```
WHILE 1 = 1

BEGIN

    DELETE TOP (2500)

    FROM sales_history WHERE order_date <= '20030101'

    IF @@rowcount < 2500 BREAK

END
```

The *TOP* clause should be used in favor of the old *SET ROWCOUNT* statement because the *TOP* clause is open to many more query optimization algorithms. (Prior to SQL Server 2005, the *SET ROWCOUNT* statement was often used with data modification transactions to process large numbers of rows in batches, thus preventing the transaction log from filling up and ensuring that row locks would not escalate to a full table lock.)

See Also

- INSERT
- SELECT
- TRUNCATE TABLE
- UPDATE

DISCONNECT Statement

The *DISCONNECT* statement terminates one or more connections created between the current SQL process and the database server.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DISCONNECT {CURRENT | ALL | connection_name | DEFAULT}
```

Keywords

CURRENT

Closes the currently active user connection.

ALL

Closes all open connections for the current user.

Rules at a Glance

DISCONNECT is used to disconnect a named SQL session (*connection_name*), the *CURRENT* connection, the *DEFAULT*

connection, or *ALL* connections held by the user. For example, we can disconnect a single session called `new_york`:

```
DISCONNECT new_york
```

In the next example, we might want to disconnect all currently open sessions for the current user process:

```
DISCONNECT ALL
```

Programming Tips and Gotchas

DISCONNECT is not universally supported across platforms. Do not build cross-platform applications based on *DISCONNECT* unless you've made provisions to disconnect SQL sessions using each platform's preferred disconnection methodology.

DB2

DB2 supports the ANSI standard for *DISCONNECT* in embedded SQL only. *DISCONNECT* can drop one or more connections only after a commit or rollback operation succeeds for the sessions to be dropped. You can also use the *RELEASE* statement to place a connection in a disconnect-pending condition.

MySQL

Not supported.

Oracle

Oracle allows *DISCONNECT* only in its ad hoc query tool, SQL*Plus, using this syntax:

```
DISC[ONNECT]
```

In this usage, the command ends the current session with the database server but otherwise allows work in SQL*Plus to continue. For example, a programmer can continue to edit the buffer, save run files, and so on. However, you must establish a new connection to issue any SQL commands. Exiting SQL*Plus and returning to the filesystem requires the *EXIT* or *QUIT* commands. For example, to end the current connection with an Oracle server:

```
DISCONNECT;
```

PostgreSQL

Not supported. Instead, every programming interface supports a disconnect operation; for example, *SPI_FINISH* is available under the Server Programming Interface, and *PG_CONNECT* is available under the PL/TCL programming package.

SQL Server

Microsoft SQL Server supports the ANSI syntax for *DISCONNECT* in Embedded-SQL (ESQL) only, not within its ad hoc querying tool, SQL Query Analyzer. To disconnect cleanly from Microsoft SQL Server in an ESQL program, use the *DISCONNECT ALL* statement.

See Also

CONNECT

DROP Statements

All of the database objects created with *CREATE* statements may be destroyed using complementary *DROP* statements. On some platforms, a *ROLLBACK* statement after a *DROP* statement will recover the dropped object. However, on other

database platforms, the *DROP* statement is irreversible and permanent, so it is advisable to use the command with care.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

Currently, the SQL2003 standard supports the ability to drop a lot of object types that are largely unsupported by most vendors. The ANSI SQL2003 syntax follows this format:

DROP {object_type} object_name {RESTRICT | CASCADE}

Keywords

DROP object_type object_name

Irreversibly and permanently destroys the specified object, called *object_name*, of the type identified. The *object_name* does not need a schema identifier, but the current schema is then assumed. ANSI SQL2003 supports a long list of object types, each created with their own corresponding *CREATE* statement. *CREATE* statements covered in this book with corresponding *DROP* statements include:

DOMAIN	SCHEMA
FUNCTION	TABLE
METHOD	TRIGGER
PROCEDURE	TYPE
ROLE	VIEW

RESTRICT | CASCADE

Prevents the *DROP* from taking place if any dependent objects exist (*RESTRICT*), or causes all dependent objects to also be dropped (*CASCADE*). This clause is not allowed with some forms of *DROP*, like *DROP TRIGGER*, but is mandatory for others, like *DROP SCHEMA*. To further explain, *DROP SCHEMA RESTRICT* will only drop an empty schema. Otherwise, if the schema contains objects, the operation will be prevented. In contrast, *DROP SCHEMA CASCADE* will drop a schema and all objects contained therein.

Rules at a Glance

For rules about the creation or modification of each of the object types, refer to the corresponding *CREATE/ALTER* statements.

The *DROP* statement destroys a pre-existing object. The object is permanently destroyed, and all users who had access to the object immediately lose the ability to access the object.

The object may be qualified — that is, you may fully specify the schema where the dropped object is located. For example:

DROP TABLE scott.sales_2004 CASCADE;

The above statement would not only drop the table **scott.sales_2004**, but also any views, triggers, or constraints built on it. On the other hand, a *DROP* statement may include an unqualified object name, and then the current schema context is assumed. For example:

```
DROP TRIGGER before_ins_emp;
```

```
DROP ROLE sales_mgr;
```

Although not required by the SQL2003 standard, most implementations cause the *DROP* command to fail if the database object is in use by another user.

Programming Tips and Gotchas

DROP will only work when it is issued against a pre-existing object of the appropriate type where the user has appropriate permissions, usually the *DROP TABLE* permission. (Refer to the [GRANT Statement](#) section for more information.) The SQL standard requires only that the owner of an object may drop it, but most database platforms allow variations on that requirement. For example, the database superuser can usually drop any object on the database server.

With some vendors, the *DROP* command fails if the database object has extended properties. For example, Microsoft SQL Server will not drop a table that is replicated unless you first remove the table from replication.



It is important to be aware that most vendors do not notify you if the *DROP* command creates a dependency problem. Thus, if a table that is used by a few views and by stored procedures elsewhere in the database is dropped, no warning is issued. Those other objects simply return failures when they are accessed. To prevent this problem, you may wish to use the *RESTRICT* syntax where it is available.

You may have noticed that the ANSI standard does not support certain common *DROP* commands like *DROP DATABASE* and *DROP INDEX*, even though every vendor covered in this book (and just about every one in the market) supports these commands. The exact syntax for each of these commands is covered in the platform-specific sections below.

DB2

DB2 does not support the *CASCADE* option. It supports many of the ANSI object types in addition to many of its own object types, as follows:

```
DROP {BUFFERPOOL | EVENT MONITOR | [SPECIFIC] FUNCTION |  
      FUNCTION MAPPING | INDEX | INDEX EXTENSION |  
      [SPECIFIC] METHOD | NICKNAME | DATABASE PARTITION GROUP |  
      PACKAGE | [SPECIFIC] PROCEDURE | SCHEMA | SEQUENCE | SERVER |  
      TABLE [HIERARCHY] | TABLESPACE[S] | TRANSFORM[S]  
      [{ALL | group_name}] FOR | TRIGGER | [DISTINCT] TYPE |  
      TYPE MAPPING | USER MAPPING FOR {authorization_name | USER} SERVER |  
      VIEW [HIERARCHY] | WRAPPER } object_name [RESTRICT]
```

The rules for DB2's *DROP* statements are less consistent than the ANSI standard's rules, so the full syntax of each *DROP* variant is shown below:

```
BUFFERPOOL bufferpool_name
```

Drops the named bufferpool as long as no tablespaces are assigned to it. Memory is recovered immediately, but disk space may be recovered at the next connection to the database.

EVENT MONITOR *event_monitor_name*

Drops the named event monitor as long as the event monitor is not set ON.

[SPECIFIC] FUNCTION *function_name* [(*datatype* [,...])] [RESTRICT]

Drops the named user-defined function. DB2 allows multiple functions of the same name. To drop these, use the syntax *DROP FUNCTION function_name (data type [,...])*, providing *data type* identifiers sufficient to uniquely identify the user-defined function. Use the optional *SPECIFIC* keyword to identify functions by the unique name assigned to each at creation time. You may drop a function while it is in use without causing the user to experience a failure (until they try to access it later). However, you *cannot* drop a function that is a component of a package, triggers, views, etc. Here are examples of all three syntaxes:

DROP FUNCTION sales_summary_1992

DROP FUNCTION sales_summary (int, float)

DROP SPECIFIC FUNCTION products.inventory_level

FUNCTION MAPPING *function_mapping_name*

Drops the named function mapping.

INDEX [EXTENSION] *index_name* [RESTRICT]

Drops the named index or, optionally, an index extension, as long as it is not a primary key or unique constraint. You should use *ALTER TABLE* to drop a primary key or unique constraint. *RESTRICT* is mandatory when dropping an index extension.

[SPECIFIC] METHOD *method_name* [(*datatype* [,...])]

[FOR *type_name*][RESTRICT]

Drops the named method body or specific method body, except those implicitly created by *CREATE TYPE*. The method specification remains as part of the definition of the subject type, unless removed with *ALTER TYPE DROP METHOD*. You may alternately use the syntax *DROP METHOD method_body_name (data type1 [,...])* to further identify the method via datatypes. The *FOR type_name* clause must be specified when *SPECIFIC* is not specified in order to name the type where the method will be dropped. All dependencies on a method must be dropped before the method can be dropped.

NICKNAME *nickname*

Drops the named nickname. Certain other dependent objects are also dropped, including materialized query tables, index specifications, views (not dropped but marked inoperable), and packages (not dropped but marked inoperable).

DATABASE PARTITION GROUP *db_prtn_name*

Drops the named database partition group, and all tablespaces and subordinant objects defined under it (such as tables, packages, constraints, etc.) are dropped or marked inoperable. You should drop any tablespaces from the database partition group first. For example:

DROP TABLESPACE sales_2004

DROP TABLESPACE sales_2003

DROP DATABASE PARTITION GROUP sales_archive_0304

PACKAGE *package_name* [VERSION *version_id*]

Drops the named package of *schema_name.package_id*. You may also optionally add the clause *VERSION version_id* after the *package_id* to further identify the package.

[SPECIFIC] PROCEDURE *procedure_name* [(*datatype1* [,...])] [RESTRICT]

Drops the named procedure or specific procedure. DB2 allows multiple procedures with the same name, each distinguished by their different datatype usage. You may alternately identify the procedure to drop via the syntax *DROP PROCEDURE procedure_name (datatype [,...])* syntax.

SCHEMA *schema_name* RESTRICT

Drops the named schema. The *RESTRICT* keyword requires that the schema be empty before it can be dropped.

SEQUENCE *sequence_name* [RESTRICT]

Drops the named sequence from the server. However, you cannot drop a sequence created for an *IDENTITY* column using the *DROP SEQUENCE* statement.

SERVER *server_name*

Drops the definition of a named data source from the catalog as well as all nicknames for objects residing on the remote server. In addition, index specifications, user-defined functions, user-defined type mappings, and user mappings dependent on the remote server are dropped. Packages and function mappings dependent on the remote server are invalidated.

TABLE [HIERARCHY] *table_name*

Drops the named base table, declared temporary table, materialized query table, or staging table from the server. Indexes, constraints, materialized query tables, and staging tables that reference a dropped table are also dropped. Subtables of a typed table are not automatically dropped and must be dropped before the parent table can be dropped. Views, packages, and triggers that reference the dropped table are rendered inoperable. Any DATALINK columns are unlinked, though asynchronously. The *HIERARCHY* option is used to drop a named type table hierarchy (i.e., both the parent table and all of its subtables). In this syntax, the **object_name** is the name of the root table in the typed table hierarchy. A table cannot be dropped if it has the *RESTRICT ON DROP* attribute.

TABLESPACE[S] *tablespace_name* [,...]

Drops one or more named tablespaces and all objects from those tablespaces, unless a table stored in the tablespace spans the tablespace to be dropped and at least one of the tablespaces spanned, or the table has the *RESTRICT ON DROP* property. The *TABLESPACE* keyword and the *TABLESPACES* keyword are synonyms.

TRANSFORM[S] {ALL | *group_name*} FOR *type_name*

Drops all transform groups for a user-defined datatype using the syntax *DROP TRANSFORM ALL FOR datatype_name*. Note that only the transform group and not the user-defined datatype are dropped. Alternately, use the syntax *DROP TRANSFORM transform_group_name FOR datatype_name* to drop a single transform group from a user-defined datatype. For example:

DROP TRANSFORM sales_calcs FOR orders

TRIGGER *trigger_name*

Drops the named trigger. This may also cause certain packages to be marked invalid.

[DISTINCT] TYPE *type_name* [RESTRICT]

Drops the named user-defined type or, with the optional *DISTINCT* keyword, drops the named distinct type. Using the optional *RESTRICT* keyword prevents the type from being dropped, if the type has any dependency such as being used in a table definition or in a view, has subtypes, etc.

TYPE MAPPING *type_mapping_name*

Drops the named user-defined type mapping, though no other subordinant objects are dropped.

USER MAPPING FOR {*user_name* | USER} SERVER *server_name*

Drops the user mapping to a federated server for a named user. If no server is specified, all user mappings are dropped for the named user. For example, the user **Dylan** no longer needs the **oracle50** data source:

```
DROP USER MAPPING FOR dylan SERVER oracle50
```

VIEW [HIERARCHY] *view_name*

Drops the named view from the schema. If the view is a parent object, then all subviews must be dropped before the parent view. Other objects that directly or indirectly depend on the view, such as triggers or materialized query tables, are marked inoperable. The optional *HIERARCHY* keyword tells DB2 to drop a parent typed view and all of its subviews.

WRAPPER *wrapper_name*

Drops the named wrapper as well as any server definitions, user-defined function mappings, and user-defined datatype mappings. This can cause a cascading action when dropping a server definition because objects dependent on the server definition (things like user mappings and nicknames) are also dropped. If the server definition results in a nickname being dropped, then objects dependent on the nickname (like views or packages) become inoperable.

RESTRICT

Optionally allowed on *FUNCTION*, *SPECIFIC FUNCTION*, *METHOD*, *SPECIFIC METHOD*, *PROCEDURE*, *SPECIFIC PROCEDURE*, *SEQUENCE*, *TYPE*, and *DISTINCT TYPE*. Mandatory for *INDEX EXTENSION* and *SCHEMA*. Like the ANSI standard, *RESTRICT* prevents the object from being dropped if a dependant object exists. The *RESTRICT* behavior is the default.

If you do not specify a schema along with the object name when performing a drop, DB2 will assume the current schema in dynamic SQL statements. DB2 will apply the *QUALIFIER* precompile/bind option for static SQL statements.

MySQL

MySQL supports a very limited number of objects via the *CREATE/ALTER* statement. Consequently, the syntax for *DROP* is also limited, as in the following:

```
DROP { DATABASE | [TEMPORARY] TABLE | INDEX } object_name [,...]
```

```
    [ON table_name]
```

```
    [IF EXISTS] object_name [,...] [RESTRICT [CASCADE]]
```

The syntax elements are:

DATABASE *database_name*

Drops the named database, including all the objects it contains, such as tables and indexes. The *DROP DATABASE* command removes all database and table files from the filesystem, as well as two-digit subdirectories. MySQL will return a message showing how many files were erased from the database directory. (Files of these extensions are erased: *.BAK*, *.DAT*, *.frm*, *.HSH*, *.ISD*, *.ISM*, *.MRG*, *.MYD*, *.MYI*, *.dm*, and *.fm*.) If the database is linked, both the link and the database are erased. You may only drop one database at a time. *RESTRICT* and *CASCADE* are not valid on *DROP DATABASE*.

[TEMPORARY] TABLE *table_name* [,...]

Drops one or more named tables, with table names separated from each other by commas. MySQL erases each table's definition and deletes the three table files from the filesystem (*.frm*, *.MYD*, and *.MYI*). Issuing this command causes MySQL to commit all active transactions. The *TEMPORARY* keyword drops only temporary tables without committing running transactions or checking access rights.

INDEX *index_name* ON *table_name*

Drops the named index from the named table on Version 3.22 or later. This command doesn't do anything on versions earlier than 3.22, but on 3.22 it actually executes an `ALTER TABLE...DROP INDEX` command behind the scenes. You cannot use `RESTRICT`, `CASCADE`, or `IF EXISTS` with a `DROP INDEX` statement.

`RESTRICT` | `CASCADE`

Acts as noise. These keywords do not generate an error by MySQL, nor do they do anything.

`IF EXISTS`

Prevents an error message when attempting to drop an object that does not exist. Usable in MySQL Version 3.22 or later.

MySQL supports *only* the ability to drop a database, a table (or tables), or an index from a table. Although the `DROP` statement will not fail with the `RESTRICT` and `CASCADE` optional keywords, they also don't do anything. You can use the `IF EXISTS` clause to prevent MySQL from returning an error message by trying to delete an object that doesn't exist.

Oracle

Oracle supports most of the ANSI keywords for the `DROP` statements, as well as many additional keywords corresponding to objects uniquely supported by Oracle, as follows:

`DROP {CLUSTER | CONTEXT | DATABASE | DATABASE LINK | DIMENSION |`

`DIRECTORY | FUNCTION | INDEX | INDEXTYPE | JAVA |`

`LIBRARY | MATERIALIZED VIEW | MATERIALIZED VIEW LOG |`

`OPERATOR | OUTLINE | PACKAGE | PROCEDURE | PROFILE |`

`ROLE | ROLLBACK SEGMENT | SEQUENCE | SYNONYM | TABLE |`

`TABLESPACE | TRIGGER | TYPE | TYPE BODY | USER | VIEW }`

object_name

The rules for Oracle `DROP` statements are less consistent than the ANSI standard's rules, so the full syntax of each `DROP` variant is shown below:

CLUSTER *cluster_name* [INCLUDING TABLES [CASCADE CONSTRAINTS]]

Drops the named cluster from the database. (Note that this drops a cluster; it does not uncluster a specific table from a cluster.) `INCLUDING TABLES` drops all tables that belong to the cluster. `CASCADE CONSTRAINTS` drops all referential integrity constraints that refer to the tables in the cluster. `DROP CLUSTER` will fail without `CASCADE CONSTRAINTS` when tables outside of the cluster have constraints (like a primary or foreign key) that refer to tables in the cluster. For example:

`DROP CLUSTER hr INCLUDE TABLES CASCADE CONSTRAINTS;`

CONTEXT *namespace*

Drops the named context from the database. Users currently in the namespace are unaffected, but will be unable to access the context when they next attempt to use that context. For example:

`DROP CONTEXT hr_context`

DATABASE

Drops a database from the Oracle server.

[PUBLIC] DATABASE LINK *database_link_name*

Drops the named database link from the database. You must include the *PUBLIC* keyword to drop a public database link. Note that the database link cannot be schema qualified. For example, to drop the database link called *hq.southeast.tenn*:

```
DROP DATABASE LINK hq.southeast.tenn;
```

DIMENSION *dimension_name*

Drops the named dimension. This does not invalidate materialized views built on the dimension, though they may execute more slowly and query requests may be rewritten.

DIRECTORY *directory_name*

Drops the named directory object from the database. This should not be used on directories containing files accessed by PL/SQL or OCI programs. For example, you could drop the *image_files* directory:

```
DROP DIRECTORY image_files;
```

FUNCTION *function_name*

Drops the named function, as long as it is not a component of a package. If you want to drop a function from a package, use the *CREATE PACKAGE...OR REPLACE* statement to redefine the package without the function you want to be rid of. Any local objects that depend on or call the function are invalidated and any statistical types associated with the function are disassociated.

INDEX *index_name* [FORCE]

Drops a named index or domain index from the database. Dropping an index invalidates all objects that depend on the parent table, including views, packages, functions, and stored procedures. Dropping an index invalidates cursors and execution plans that use the index, and will force a hard parse of the affected SQL statements when they are next executed.

Non-IOT indexes are secondary objects and can be dropped and recreated without any loss of user data. IOTs, because they combine both table and index data in the same structure, cannot be dropped and recreated in this manner. IOTs should be dropped using the *DROP TABLE* syntax.

When you drop a partitioned index, all partitions are dropped. When you drop a composite-partitioned index, all index partitions and subpartitions are dropped. When you drop a domain index, any statistics associated with the domain index are removed and any statistic types are disassociated. The optional keyword *FORCE* applies only when dropping domain indexes. *FORCE* allows you to drop a domain index marked *IN PROGRESS*, or to drop a domain index when its *indextype* routine invocation returns an error. For example:

```
DROP INDEX ndx_sales_salesperson_quota;
```

INDEXTYPE *indextype_name* [FORCE]

Drops the named *indextype* along with any associated statistic types. Oracle also drops any statistics collected with the statistics type. The optional keyword *FORCE* drops the *indextype* even when it is currently referenced by one or more domain indexes and marks the applicable domain indexes as *INVALID*. If you do not use *FORCE* on an *indextype* that is referenced by one or more domain indexes, the statement will fail. For example:

```
DROP INDEXTYPE imageindextype FORCE;
```

JAVA {SOURCE | CLASS | RESOURCE} *"java_object_name"*

Drops the named Java source, Java class, or Java resource schema object. *SOURCE* drops the named Java source schema object and all Java class schema objects derived from it. *CLASS* drops the named Java class schema object. *RESOURCE* drops the named Java resource schema object. For example, to drop the Java source *MyJavaSource* and all dependent schema objects:

```
DROP JAVA SOURCE "MyJavaSource";
```

LIBRARY *library_name*

Drops the named external procedure library.

{MATERIALIZED VIEW | SNAPSHOT} *mview_name*

Drops the named materialized view. You can use the keyword *SNAPSHOT* as an alternative to *MATERIALIZED VIEW* for backward compatibility. You must have privileges to drop the table plus views and indexes that make up the materialized view. Note that if you drop a parent table, any materialized views built upon it are not automatically dropped. But Oracle will return an error when a refresh is attempted on a materialized view whose parent table has been dropped. For example, we can drop a materialized view called **salespeople_at_quota**:

```
DROP MATERIALIZED VIEW salespeople_at_quota;
```

{MATERIALIZED VIEW | SNAPSHOT} LOG ON *parent_table_name*

Drops the named materialized view log that resides on the parent table (also known as the master table). You can use the statement *DROP SNAPSHOT* as an alternative to *DROP MATERIALIZED VIEW* for backward compatibility. RowID, primary key, and subquery materialized views can no longer be *fast refreshed* after the materialized view log has been dropped.

OPERATOR *operator_name* FORCE

Drops the named user-defined operator from the database. The optional keyword *FORCE* allows you to drop the operator even if it is referenced by other schema objects such as triggers, packages, user-defined functions, and so forth. In that case, the dependent objects become invalid. Dropping a user-defined operator with dependents without the *FORCE* keyword will result in an error.

OUTLINE *outline_name*

Drops the named stored outline from the database. The optimizer will generate new query plans for any SQL statements that used the stored outline after it is dropped.

PACKAGE [BODY] *package_name*

Drops the named package body and specification. The optional keyword *BODY* drops only the body of the package, but leaves the specification unaltered. This option is useful if you do not want to invalidate dependent objects, though you cannot call any components of the package until the package body is recreated. Otherwise, if you drop both body and specification, Oracle invalidates all dependent objects. (Note that if you want to drop a single object from a package, you should use the statement *CREATE PACKAGE [BODY]...OR REPLACE*.) The following examples drop first the package body and specification of the **sales_quota_calc** package, and then drop only the body of the **regional_sales_calc** package:

```
DROP PACKAGE sales_quota_calc;
```

```
DROP PACKAGE BODY regional_sales_calc;
```

PROCEDURE *procedure_name*

Drops the named stored procedure. Any dependent objects are invalidated when you drop a stored procedure. Dependent objects will fail with an error if they are accessed before you recreate the stored procedure. If you recreate the stored procedure, and a dependent object is accessed, the dependent object will be recompiled.

PROFILE *profile_name* [CASCADE]

Drops the named profile. The optional keyword *CASCADE* removes the profile from any users who were granted the profile. *CASCADE* is only required if the profile has been granted to one or more users. Users who no longer have a profile are then assigned the DEFAULT password. For example, the following statement drops the **web_user** profile and removes it from all users:

```
DROP PROFILE web_user CASCADE;
```

ROLE *role_name*

Drops the named role, removes it from the database, and revokes it from all users and roles where it has been granted. No new sessions can use the role, but sessions that are currently running under the role are not affected. For example, the following statement drops the **sales_mgr** role:

```
DROP ROLE sales_mgr;
```

ROLLBACK SEGMENT *rbseg_name*

Drops the named rollback segment and returns its space to the tablespace where the rollback segment was assigned. The rollback segment must be on-line and may not be the SYSTEM rollback segment. (Note that in Automatic Undo Management mode, you can only drop rollback segments. You cannot create or alter them.) For example, the following statement drops the rollback segment named **rbseg_sales_tbspc**:

```
DROP ROLLBACK SEGMENT rbseg_sales_tbspc;
```

SEQUENCE *sequence_name*

Drops the named sequence. (Note that the only way to restart a sequence is to drop it and recreate it.)

[PUBLIC] SYNONYM *synonym_name* [FORCE]

Drops the named private or *PUBLIC* synonym. (Note that the only way to redefine a synonym is to drop it and recreate it.) You cannot specify a schema when dropping a *PUBLIC* synonym. Materialized views become invalid when you drop their synonym. The optional keyword *FORCE* allows you to drop a synonym even if it has dependent tables or user-defined types. Otherwise, you cannot drop a synonym with dependent tables or user-defined types. Oracle discourages the use of *FORCE*. For example, you could drop the public synonym called **users**:

```
DROP PUBLIC SYNONYM users;
```

TABLE *table_name* [CASCADE CONSTRAINTS] [PURGE]

Drops the named table, erases all of its data, drops all indexes and triggers built from the table (even those in other schemas), and invalidates all permissions and all dependent objects (like views, stored procedures, etc.). On partitioned tables, Oracle drops all their partitions (and subpartitions). On index-organized tables, Oracle drops all dependent mapping tables. Statistic types associated with a dropped table are disassociated. Materialized view logs built on a table are also dropped when the table is dropped.

The *DROP TABLE* statement is effective for standard tables, index-organized tables, and object tables. The table being dropped is only moved to the recycling bin, unless you add the optional keyword *PURGE*, which tells Oracle to immediately free all space consumed by the table. (Oracle also supports a non-ANSI SQL command called *PURGE* that lets you remove tables from the recycling bin outside of the *DROP TABLE* statement.) *DROP TABLE* erases only the metadata of an external table. You must use an external operating system command to drop the file associated with an external table and reclaim its space.

Use the optional *CASCADE CONSTRAINTS* clause to drop all referential integrity constraints elsewhere in the database that depend on the primary or unique key of the dropped table. You cannot drop a table with dependent referential integrity constraints without using the *CASCADE CONSTRAINTS* clause. The following example drops the **job_desc**

table in the **emp** schema, then drops the **job** table and all referential integrity constraints that depend on the primary key and unique key of the **job** table:

```
DROP TABLE emp.job_desc;
```

```
DROP TABLE job CASCADE CONSTRAINTS;
```

TABLESPACE *tablespace_name* [INCLUDING CONTENTS [AND DATAFILES] [CASCADE CONSTRAINTS]

Drops the named tablespace (preferably when it is offline), deleting all datafiles and tempfiles. Oracle-managed datafiles and tempfiles in the tablespace are deleted from the filesystem, but datafiles and tempfiles managed by the OS can also be deleted by including the optional clause *INCLUDING CONTENTS AND DATAFILES*. Note that you cannot drop a tablespace when it contains a domain index, objects created by a domain index, data needed to roll back uncommitted transactions, or the *SYSTEM* tablespace. The optional clause *INCLUDING CONTENTS* drops all contents of the tablespace, and is required for any tablespace that contains database objects. The optional clause *AND DATAFILES* also drops associated operating system files. The optional clause *CASCADE CONSTRAINTS* drops all external referential integrity constraints that depend on the primary or unique keys of tables within the tablespace and is required when such dependencies exist. The first of the following examples drops the tablespace **sales_tbspc** and all dependent referential integrity constraints, while the second example drops the tablespace **sales_images_tbspc** and all associated operating system files:

```
DROP TABLESPACE sales_tbspc
```

```
INCLUDING CONTENTS CASCADE CONSTRAINTS;
```

```
DROP TABLESPACE sales_image_tbspc
```

```
INCLUDING CONTENTS AND DATAFILES;
```

TRIGGER *trigger_name*

Drops the named trigger from the database.

TYPE [BODY] *type_name* [{FORCE | VALIDATE}]

Drops the specification and body of the named object type, nested table type, or *VARRAY*, as long as they have no type or table dependencies. You must use the optional *FORCE* keyword to drop a supertype, a type with an associated statistics type, or a type with any sort of dependencies. All subtypes and statistic types are then invalidated. Oracle will also drop any public synonyms associated with a dropped type. The optional *BODY* keyword tells Oracle to drop only the body of the type while keeping its specification intact. Use the optional *VALIDATE* keyword when dropping subtypes to check for stored instances of the named type in any of its subtypes. Oracle then performs the drop only if no stored instances are found. For example:

```
DROP TYPE salesperson_type;
```

USER *user_name* [CASCADE]

Drops the named user from the database. By default, Oracle will not drop a user whose schema still owns objects. The optional keyword *CASCADE* tells Oracle to also remove all of the user's objects, as well as applicable dependent objects. Oracle does not drop objects from other users' schemas that reference those in the dropped schema, though it does invalidate them. For example, the following statement drops the user **cody** and all of his objects:

```
DROP USER cody CASCADE;
```

VIEW *view_name* [CASCADE CONSTRAINTS]

Drops the named view and marks as invalid any views, subviews, synonyms, or materialized views that refer to the dropped view. Use the optional clause *CASCADE CONSTRAINTS* to drop all referential integrity constraints that depend on the view. Otherwise, the *DROP* statement will fail if dependent referential integrity constraints exist. For example, the following statement drops the **active_employees** view in the **hr** schema:

```
DROP VIEW hr.active_employees;
```

In all of the syntax shown above, *object_name* can be replaced with [*schema_name.*]*object_name*. If you leave off the schema name, the default schema of the user session is assumed. Thus, the following *DROP* statement drops the specified view from the **sales_archive** schema:

```
DROP VIEW sales_archive.sales_1994;
```

However, if your personal schema is **scott**, then the following command is assumed to be against **scott.sales_1994**:

```
DROP VIEW sales_1994;
```

PostgreSQL

PostgreSQL does not support the *RESTRICT* or *CASCADE* optional keywords supported by the ANSI standard. It supports a wide variety of *DROP* variants, as follows:

```
DROP { AGGEGATE | DATABASE | FUNCTION | GROUP | INDEX |  
      LANGUAGE | OPERATOR | RULE | SEQUENCE | TABLE | TRIGGER |  
      TYPE | USER | VIEW } object_name
```

Following is the full syntax for each variant:

```
AGGEGATE aggregate_name ( { datatype | * } )
```

Drops the user-defined aggregate function. The value for *datatype* is the datatype of the existing aggregate function, but you may substitute an asterisk (***) to match any input datatype. For example, we can drop an aggregate function called **sales_avg** we made using PostgreSQL's *CREATE AGGREGATE* statement:

```
DROP AGGREGATE sales_avg(int4);
```

```
DATABASE database_name
```

Drops the named database and erases the operating system directory containing all of the database's data. This command can only be executed by the database owner while connected to a database other than the target database. For example, we can drop the **sales_archive** database:

```
DROP DATABASE sales_archive;
```

```
FUNCTION function_name ( [datatype1 [,...]] )
```

Drops the named user-defined function. Since PostgreSQL allows multiple functions of the same name distinguished only by the various input parameters they require, you must specify one or more datatypes to uniquely identify the user-defined function you wish to drop. PostgreSQL does not perform any kind of dependency checks on other objects that might reference a dropped user-defined function. (They will fail when invoked against an object that no longer exists.) For example:

```
DROP FUNCTION median_distribution (int, int, int, int);
```

```
GROUP group_name
```

Drops the named group, though it does not affect users who were in the group. This is similar to *DROP ROLE* on

other database platforms.

INDEX *index_name* [,...]

Drops one or more named indexes that you own. For example:

```
DROP INDEX ndx_titles, ndx_authors;
```

[PROCEDURAL] LANGUAGE *language_named*

Drops the named user-defined procedural language. You can optionally enclose the name in single quotes. Note that PostgreSQL does not check to see if any user-defined functions or triggers written in the named language still exist in the database. For example:

```
DROP LANGUAGE php;
```

OPERATOR *operator_id* ({*left_datatype* | NONE }, {*right_datatype* | NONE })

Drops a named user-defined operator where the value of *left_datatype* is the operator's left argument and *right_datatype* is the operator's right argument. *NONE* may be substituted for either argument when there is no argument in that position. (PostgreSQL does not check for any dependent access methods or operator classes that rely on the dropped operator. You should check for them and drop them manually to avoid errors when they are invoked.) For example, the first of the following statements drops the power operator, and the second drops the unary negation operator:

```
DROP OPERATOR ^ (int4, int4);
```

```
DROP OPERATOR ! (NONE, bool);
```

RULE *rule_name* [,...]

Drops one or more named rules from the database. PostgreSQL then immediately stops enforcing the rules and erases historical data about the rules. For example:

```
DROP RULE phone_rule;
```

SEQUENCE *seq_name* [,...]

Drops one or more named sequences from the database. Since the current version of PostgreSQL treats sequences as special tables, the *DROP SEQUENCE* statement works very much like the *DROP TABLE* statement.

TABLE *table_name* [,...]

Drops one or more existing tables from the database, as well as any indexes or triggers specified for the table. For example, you can drop more than one table in a single statement:

```
DROP TABLE authors, titles;
```

TRIGGER *trigger_name* ON *table_name*

Drops the named trigger from the database. Since PostgreSQL requires that trigger names be unique only on the table they are attached to, it is possible to have many triggers called, say, **insert_trigger** or **delete_trigger**, each on a different table. For example:

```
DROP TRIGGER insert_trigger ON authors;
```

TYPE *type_name* [,...]

Drops one or more pre-existing user-defined types from the database. PostgreSQL does not check to see what impact the *DROP TYPE* command might have on any dependent objects like functions, aggregates, or tables. You must check dependent objects manually. (Do *not* remove any of the built-in types that ship with PostgreSQL!) Note that PostgreSQL's implementation of types differs from the ANSI standard. Refer to the [CREATE/ALTER TYPE Statement](#) section for more information.

USER *user_name*

Drops the named user from the current database. This command does not drop any objects owned by the user, such as tables or views, but will fail if the user owns any databases. For example:

```
DROP USER emily;
```

VIEW *view_name* [,...]

Drops one or more pre-existing views from the database.

Note that PostgreSQL drop operations do not allow you to specify the target database where the operation will take place (except for *DROP DATABASE*). Therefore, you should execute any drop operation from the database where the object you want to drop is located.

SQL Server

SQL Server supports several standard variants of the *DROP* statement. SQL Server's main difference compared to the ANSI *DROP* statement is the ability to drop multiple objects with a single statement:

```
DROP { DATABASE | DEFAULT | FUNCTION | INDEX | PROCEDURE |  
      RULE | STATISTICS | TABLE | TRIGGER | TYPE | VIEW } object_name
```

Following is the full syntax for each variant:

DATABASE *database_name* [,...]

Drops the named database(s) and erases all disk files used by the database(s). This command may only be issued from the **master** database. Replicated databases must be removed from their replication schemes before they can be dropped, as must log shipping databases. You cannot drop a database while it is in use, nor can you drop system databases (**master**, **model**, **msdb**, or **tempdb**). For example, we can drop the **northwind** and **pubs** databases with one command:

```
DROP DATABASE northwind, pubs
```

```
GO
```

DEFAULT *default_name* [,...]

Drops one or more user-defined defaults from the current database. Note that this does not drop *DEFAULT* constraints, only user-defined defaults created with the *CREATE DEFAULT* statement. (You can drop default constraints using the statement *ALTER TABLE...DROP CONSTRAINT*.) When dropping a default from a column, null values will be inserted into columns that accept null. Otherwise, if a column does not accept nulls, an error is raised and no values are entered into the column. If a column or user-defined datatype being dropped is bound, then you must first unbind it using the SQL Server system stored procedure **sp_unbinddefault**. In the following two examples, we first drop a user-defined default and then unbind and drop the user-defined default:

```
DROP DEFAULT sales_date_default
```

```
GO
```

```
EXEC sp_unbindefault
```

```
'sales.ord_date'
```

```
DROP DEFAULT order_date_default
```

```
GO
```

```
FUNCTION [schema.]function_name [,...]
```

Drops one or more user-defined functions from the current database.

```
INDEX table_or_view_name.index_name [,...]
```

Drops one or more indexes from tables or indexed views in the current database and returns space to the database. This statement should not be used to drop a *PRIMARY KEY* or *UNIQUE* constraint. Instead, drop these constraints using the *ALTER TABLE...DROP CONSTRAINT* statement. When dropping a clustered index from a table, all non-clustered indexes are rebuilt. When dropping a clustered index from a view, all non-clustered indexes are dropped.

```
PROCEDURE procedure_name [,...]
```

Drops one or more stored procedures from the current database. SQL Server allows multiple versions of a single procedure via a version number, but these versions cannot be dropped individually. You must drop all versions of a stored procedure at once. System procedures (those with an **sp_** prefix) are dropped from the **master** database if they are not found in the current user database. For example:

```
DROP PROCEDURE calc_sales_quota
```

```
GO
```

```
RULE rule_name [,...]
```

Drops one or more user-defined rules from the current database. Existing data in columns governed by a rule are not checked when the rule is dropped. You may need to unbind the rule first using the SQL Server system stored procedure **sp_unbindefault**. In the following two examples, we first drop a user-defined rule, and then unbind and drop the user-defined rule:

```
DROP RULE phone_rule
```

```
GO
```

```
EXEC sp_unbindrule 'authors.phone'
```

```
DROP RULE phone_rule
```

```
GO
```

```
STATISTICS table_or_view_name.statistics_name [,...]
```

Drops statistics on one or more collections from a table or indexed view in the current database. Dropping statistics can have a big impact on query performance, since they help the query optimizer choose the best query plan. (Statistics are created with the *CREATE INDEX* and *CREATE STATISTICS* statements. You can update statistics with the *UPDATE STATISTICS* statement or retrieve information about them with the statement *DBCC SHOW_STATISTICS*.) For example, we'll create statistics on two columns in two separate tables and then

drop the statistics:

```
CREATE STATISTICS title_stats ON sales (title_id, ord_num)

WITH SAMPLE 30 PERCENT

GO

CREATE STATISTICS zip_stats ON stores (zip)

WITH FULLSCAN

GO

DROP STATISTICS sales.title_stats, stores.zip_stats

GO
```

TABLE *table_name*

Drops a named table and all data, permissions, indexes, triggers, and constraints specific to that table. (The *table_name* may be a fully qualified table name like **pubs.dbo.sales** or a simple table name, like **sales**, if the current database and owner are correct.) Views, functions, and stored procedures that reference the table are not dropped or marked as invalid, but will return an error when their procedural code encounters the missing table. Be sure to drop these yourself! You cannot drop a table referenced by a *FOREIGN KEY* constraint without first dropping the constraint. Similarly, you cannot drop a table used in replication without first removing it from the replication scheme. Any user-defined rules or defaults are unbound when the table is dropped. They must be rebound if the table is recreated.

TRIGGER *trigger_name* [,...]

Drops one or more triggers from the current database.

TYPE *type_name* [,...]

Drops one or more user-defined types from the current database.

VIEW *view_name* [,...]

Drops one or more views from the database, including indexed views, and returns all space to the database.

See Also

- CALL
- CONSTRAINTS
- CREATE/ALTER FUNCTION/PROCEDURE/METHOD
- CREATE SCHEMA
- CREATE/ALTER TABLE
- CREATE/ALTER VIEW
- DELETE
- DROP
- GRANT
- INSERT
- RETURN
- SELECT
- SUBQUERY
- UPDATE

EXCEPT Set Operator

The *EXCEPT* set operator retrieves the result sets of two or more queries including all the records retrieved by the first query that are not also found in subsequent queries. Whereas *JOIN* clauses are used to return the rows of two or more queries that are in common, *EXCEPT* is used to filter out the records that are present in only one of multiple, but similar, tables.

EXCEPT is in a class of keywords called a *set operator*. Other set operators include *INTERSECT* and *UNION*. (*MINUS* is Oracle's equivalent to the *EXCEPT* keyword. *EXCEPT* is the ANSI standard.) All set operators are used to simultaneously manipulate the result set of two or more queries, hence the term "set operators."

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Not supported

SQL2003 Syntax

There are technically no limits to the number of queries that you may combine with the *EXCEPT* operator. The general syntax is:

```
{SELECT statement1 | VALUES (expr1 [,...]) }
```

```
EXCEPT [ALL | DISTINCT]
```

```
[CORRESPONDING [BY (column1, column2, ...)] ]
```

```
{SELECT statement2 | VALUES (expr2 [,...]) }
```

```
EXCEPT [ALL | DISTINCT]
```

```
[CORRESPONDING [BY (column1, column2, ...)] ]
```

...

Keywords

VALUES (*expr1* [,...])

Generates a derived result set with explicitly declared values as *expression1*, *expression2*, etc. It is essentially a *SELECT* statement result set without the *SELECT...FROM* syntax. This is known as a *row constructor*, since the rows of the result set are manually constructed. According to the ANSI standard, multiple, hand-coded rows in a row constructor must be enclosed in parentheses and separated by commas.

EXCEPT

Determines which rows will be excluded from the single result set.

ALL

Considers duplicate rows from all result sets.

DISTINCT

Drops duplicate rows from all result sets prior to the *EXCEPT* comparison. Columns containing a NULL value are considered duplicates. (If neither *ALL* nor *DISTINCT* are used, *DISTINCT* behavior is the default.)

CORRESPONDING [BY] (*column1*, *column2*, ...)

Specifies that only columns with the listed names are returned, even if one or both queries use the asterisk shortcut.

Rules at a Glance

There is only one significant rule to remember when using *EXCEPT*:

The order, number, and datatypes of columns should be of the same category in all queries

The datatypes do not have to be identical, but they must be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest datatype size of each column in each ordinal position. For example, a query retrieving from columns containing a *VARCHAR(10)* and a *VARCHAR(15)* column will use the *VARCHAR(15)* datatype and size.

Programming Tips and Gotchas

None of the platforms support the *CORRESPONDING [BY column1, column2, ...]* clause.



On platforms that do not support *EXCEPT*, you might substitute a *NOT IN* subquery. However, *NOT IN* subqueries have different NULL handling and, on some database platforms, produce different result sets.

According to the ANSI standard, the *UNION* and *EXCEPT* set operators evaluate with equal precedence. However, the *INTERSECT* set operator evaluates first before the other set operators. We recommend that you explicitly control the precedence of set operators, using parentheses as a general best practice.

According to the ANSI standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. In order to avoid column and table ambiguity, be sure to alias each column for each table with the same respective alias. For example:

```
SELECT au_lname AS 'lastname', au_fname AS 'firstname'  
  
FROM authors  
  
EXCEPT  
  
SELECT emp_lname AS 'lastname', emp_fname AS 'firstname'  
  
FROM employees  
  
ORDER BY lastname, firstname
```

Also, while each of your column lists may list columns with correspondingly compatible datatypes, there may be some variation in behavior across the DBMS platforms with regard to varying length of the columns. For example, if the **au_lname** column in the previous example's first query is markedly longer than the **emp_lname** column in the second query, each platform may apply different rules as to which length is used for the final result. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the names vary across column lists. In general, the column names of the first query are used.

DB2

DB2 supports the *EXCEPT* and *EXCEPT ALL* keywords of the ANSI standard, plus the *VALUES* extension:

```
{ SELECT statement1 | VALUES (expression1, expression2[,...]) }  
  
EXCEPT [ALL]  
  
{ SELECT statement2 | VALUES (expression1, expression2[,...]) }  
  
EXCEPT [ALL]  
  
...
```

where:

VALUES

Allows you to specify one or more hand-coded rows to include in the final result set. (This is known as a *row constructor*.) The *VALUES* clause must specify the exact same number of columns as the queries in the *EXCEPT*.

Although *EXCEPT DISTINCT* is not supported, *EXCEPT* is the functional equivalent. The *CORRESPONDING* clause is not supported. In addition, the datatypes *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK*, and the structured types are not usable with *EXCEPT*, but are useable with *EXCEPT ALL*.

When the result set has the same column name for a given column across all *SELECT* statements, then that name is used as the final column name returned by the statement. If, however, the given column is named differently in different *SELECT* statements, then you must rename the column in each query using the same *AS alias_name*.

If multiple set operations are used in a single query, those enclosed in parentheses are evaluated first. Then, the queries are evaluated in left-to-right order. However, all *INTERSECT* operations are performed before any *UNION* and *EXCEPT* operations.

For example:

```
(SELECT empno
  FROM employee
  WHERE workdept LIKE 'E%'
EXCEPT
SELECT empno
  FROM emp_act
  WHERE projno IN ('IF1000', 'IF2000', 'AD3110') )
UNION
VALUES ('AA0001'), ('AB0002'), ('AC0003')
```

The example above retrieves all IDs for all employees from the **employee** table who are in any department that starts with "E," then it excludes any employee IDs from the employee accounts (**emp_act**) table representing employees working in the projects **IF1000**, **IF2000**, and **AD3110**. Finally, it adds in three extra employee IDs — **AA0001**, **AB0002**, and **AC0003** using the *UNION* set operator.

MySQL

EXCEPT is not supported in MySQL. However, you can use the *NOT IN* or *NOT EXISTS* operations as alternatives.

Oracle

Oracle does not support the *EXCEPT* set operator. However, it has an alternative set operator, *MINUS*, with identical functionality.

```
<SELECT statement1>
MINUS
<SELECT statement2>
MINUS
...
```

MINUS DISTINCT and *MINUS ALL* are not supported. *MINUS* is the functional equivalent of *MINUS DISTINCT*. Oracle does not support *MINUS* on queries under the following circumstances:

- Queries containing columns of *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatypes
- Queries containing the *FOR UPDATE* clause

- Queries containing *TABLE* collection expressions

If the first query in a set operation contains any expressions in the select item list, then include *AS* clauses to associate aliases with those expressions. Also, only the last query in the set operation may contain an *ORDER BY* clause.

For example, you could generate a list of all store IDs that do not have any records in the **sales** table:

```
SELECT stor_id FROM stores  
  
MINUS  
  
SELECT stor_id FROM sales
```

The *MINUS* command is functionally similar to a *NOT IN* query. The following query would retrieve the same results as the earlier query:

```
SELECT stor_id FROM stores  
  
WHERE stor_id NOT IN  
  
    (SELECT stor_id FROM sales)
```

PostgreSQL

PostgreSQL supports the *EXCEPT* and *EXCEPT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>  
  
EXCEPT [ALL]  
  
<SELECT statement2>  
  
EXCEPT [ALL]  
  
...
```

PostgreSQL does not support *EXCEPT* or *EXCEPT ALL* on queries with a *FOR UPDATE* clause. PostgreSQL does not support the *CORRESPONDING* clause. *EXCEPT DISTINCT* is not supported, but *EXCEPT* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *EXCEPT* or *EXCEPT ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the right-most occurrence of *ORDER BY* or *LIMIT* will be applied to the entire set operation.

PostgreSQL evaluates *SELECT* statements in a multi-*EXCEPT* statement from top to bottom unless you use parentheses to change the evaluation hierarchy of the statements.

Normally, duplicate rows are eliminated from the two result sets, unless you add the *ALL* keyword. For example, you could find all titles in the **authors** table that have no records in the **sales** table:

```
SELECT title_id  
  
FROM authors  
  
EXCEPT ALL  
  
SELECT title_id  
  
FROM sales;
```

SQL Server

EXCEPT is not supported. However, you can use *NOT IN* or *NOT EXISTS* operations in conjunction with a correlated subquery as alternatives. The following queries are examples of how you can achieve *EXCEPT* functionality using *NOT EXISTS* and *NOT IN*:

```
SELECT DISTINCT a.city
FROM pubs..authors AS a
WHERE NOT EXISTS
  (SELECT *
   FROM pubs..publishers AS p
   WHERE a.city = p.city)
SELECT DISTINCT a.city
FROM pubs..authors AS a
WHERE a.city NOT IN
  (SELECT p.city
   FROM pubs..publishers AS p
   WHERE p.city IS NOT NULL)
```

In general, *NOT EXISTS* is faster than *NOT IN*. In addition, there is a subtle issue with NULLs that differentiates the *IN* and *NOT IN* operators and the *EXISTS* and *NOT EXISTS* set operator. In order to get around this different handling of NULLs, simply add the *IS NOT NULL* clause to the *WHERE* clause, as we have in the preceding example.

See Also

```
INTERSECT
SELECT
UNION
```

EXISTS Operator

The *EXISTS* operator tests a subquery for the existence of rows.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
SELECT ...
WHERE [NOT] EXISTS (subquery)
```

The parameters and keywords are as follows:

WHERE [NOT] EXISTS

Tests the subquery for the existence of one or more rows. If even one row satisfies the subquery clause, it returns a Boolean TRUE value. The optional *NOT* keyword returns a Boolean TRUE value when the subquery

returns no matching rows.

subquery

Retrieves a result set based on a fully formed subquery.

Rules at a Glance

The *EXISTS* operator checks a subquery for the existence of one or more records against the records in the parent query.

For example, we want to see if we have any jobs where no employee is filling the position:

```
SELECT *  
FROM jobs  
WHERE NOT EXISTS  
    (SELECT * FROM employee  
     WHERE jobs.job_id = employee.job_id)
```

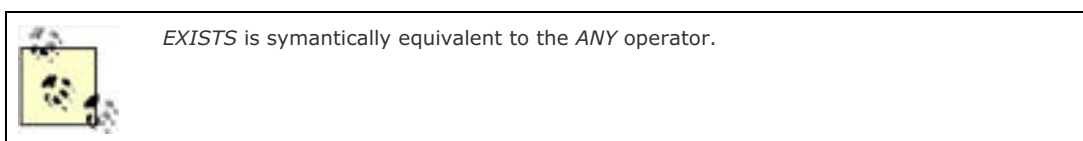
This example tests for the absence of records in the subquery using the optional *NOT* keyword. The next example looks for specific records in the subquery to retrieve the main result set:

```
SELECT au_lname  
FROM authors  
WHERE EXISTS  
    (SELECT *  
     FROM publishers  
     WHERE authors.city = publishers.city)
```

This query returns the last name of authors who live in the same city as their publishers. Note that the asterisk in the subquery is acceptable, since the subquery only needs to return a single record to provide a Boolean TRUE value. Columns are irrelevant in these cases. The first example selects only a single column. The key point is whether a *row* exists.

Programming Tips and Gotchas

EXISTS, in many queries, does the same thing as *ANY*. *EXISTS* is usually most effective with correlated subqueries.



The *EXISTS* subquery usually searches for only one of two things. Your first option is to use the asterisk wildcard (e.g., *SELECT * FROM...*) so that you are not retrieving any specific column or value. In this case, the asterisk means "any column." The second option is to select only a single column in the subquery (e.g., *SELECT au_id FROM...*). Some individual database platforms allow a subquery against more than one column (e.g., *SELECT au_id, au_lname FROM...*). However, this feature is rare and should be avoided on code that needs to be transportable across platforms.

Platform Differences

All of the platforms support *EXISTS* in the manner described above.

See Also

ALL/ANY/SOME
SELECT
WHERE

***FETCH* Statement**

The *FETCH* statement is one of four commands used in cursor processing, along with *DECLARE*, *OPEN*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than as a complete set. *FETCH* positions a cursor on a specific row and retrieves that row from the result set.

Cursors are especially important in relational databases because they are set-based, while most client-centric programming languages are row-based. So cursors allow you to perform operations a single row at a time, to better fit what a client program can do, rather than operate on a whole set of records at once.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |  
        { ABSOLUTE | RELATIVE int } }  
FROM ] cursor_name  
[INTO variable1 [, ...] ]
```

Keywords

NEXT

Tells the cursor to return the record immediately following the current row, and increments the current row to the row returned. *FETCH NEXT* is the default behavior for *FETCH*. It retrieves the first record if *FETCH* is performed as the first fetch against a cursor.

PRIOR

Tells the cursor to return the record immediately preceding the current row and decrements the current row to the row returned. *FETCH PRIOR* does not retrieve a record if it is performed as the first fetch against the cursor.

FIRST

Tells the cursor to return the first record in the cursor, making it the current row.

LAST

Tells the cursor to return the last record in the cursor, making it the current row.

ABSOLUTE int

Tells the cursor to return the *int* record from the cursor record set counting from the top (if *int* is a positive integer), or the *int* record counting from the bottom (if *int* is a negative integer), making the returned record the new current record of the cursor. If *int* is 0, no rows are returned. If the value of *int* moves the cursor past the end of the cursor result set, then the cursor is positioned after the last row (for a positive *int*) or before the first row (for a negative *int*).

RELATIVE int

Tells the cursor to return the record *int* rows after the current record (if *int* is positive) or *int* rows before the current record (if *int* is negative), making the returned record the new current row of the cursor. If *int* is 0, the current row is returned. If the value of *int* moves the cursor past the end of the cursor result set, then the cursor is positioned after the last row (for a positive *int*) or before the first row (for a negative *int*).

[FROM] cursor_name

Tells the name of the *OPEN* cursor where you want to retrieve rows. The cursor must be previously created and instantiated using the *DECLARE* and *OPEN* clauses. *FROM* is optional, but encouraged.

INTO variable1[, ...]

Stores data from each column in the open cursor into a local variable. Each column in the cursor must have a corresponding variable of a matching datatype in the *INTO* clause. Each column value is directly related to the variables in ordinal positions.

Rules at a Glance

At the highest level, a cursor must be:

1. Created using *DECLARE*
2. Opened using *OPEN*
3. Operated against using *FETCH*
4. Dismissed using *CLOSE*

By following these steps, you create a result set similar to that of a *SELECT* statement, except that you can operate against each individual row within the result set separately. For example, assume you've created and opened a cursor on a DB2 database called **employee_cursor** containing three columns:

```
DECLARE employee_cursor CURSOR FOR
```

```
SELECT lname, fname, emp_id
```

```
FROM employee
```

```
WHERE hire_date >= '2005-02-15';
```

```
OPEN employee_cursor;
```

You can now retrieve values from the cursor using *FETCH*:

```
FETCH FROM employee_cursor
```

```
INTO :emp_last_name, :emp_first_name, :emp_id
```

The *FETCH* statement allows you to position the current row within the cursor. Since you have not yet accessed the cursor in this example, you will be given the first record in the result set. You can then use any procedural code you want to operate against the values in that row.



Each database platform has its own rules about how variables are used. For example, DB2 requires a colon as a prefix on each variable, as shown in the earlier example; SQL Server requires an at sign (@), PostgreSQL and Oracle have no prefix, and so forth. Meanwhile, the SQL standard says that a colon prefix is necessary for languages that are embedded, like C or COBOL, but no prefix is needed for procedural SQL.

A cursor rests either directly on a row, before the first row, or after the last row. When a cursor is resting directly in a row, then that row is known as the *current row*. You can use cursors to position an *UPDATE* or *DELETE* statement in the current row using the *WHERE CURRENT OF* clause.

It is important to remember that a cursor result set does not wrap around. Thus if you have a result set containing 10 records and tell the cursor to move forward 12 records, it will not wrap back around to the beginning and continue its count from there. Instead, the default cursor behavior is to stop after the last record in the result set when scrolling forward, and to stop before the first record when scrolling back. For example, on SQL Server:

```
FETCH RELATIVE 12 FROM employee_cursor  
  
INTO @emp_last_name, @emp_first_name, @emp_id
```

When fetching values from the database into variables, make sure that the datatypes and number of columns and the variables match. Otherwise, you'll get an error. For example:

```
FETCH PRIOR FROM employee_cursor  
  
INTO @emp_last_name, @emp_id
```

The above example will fail since the **employee_cursor** contains three columns while the fetch operation has only two variables.

Programming Tips and Gotchas

The most common errors encountered with the *FETCH* statement are a mismatch of variables to the values in the cursor, whether that is in the exact number, order, or datatype. So take a little time to make sure you remember exactly what values are in the cursor and what their datatypes are.

Typically, the database will lock at least the current row and possibly all rows held by the cursor. According to the ANSI standard, cursor locks are not held through *ROLLBACK* or *COMMIT* operations, although this varies from platform to platform.

Although the *FETCH* statement is detailed in isolation here, it should always be managed as a group with the *DECLARE*, *OPEN*, and *CLOSE* statements. For example, every time you open a cursor, the server consumes memory. If you forget to close your cursors, you could create memory management problems. So make sure that every declared and opened cursor is eventually closed.

Cursors are also often used in stored procedures or batches of procedural code. The reason for this is because sometimes you need to perform actions on individual rows rather than on entire sets of data at a time. But because cursors operate on individual rows and not sets of data, they are often much slower than other means of accessing your data. Just make sure that you perform strong analysis on your approach. Many challenges, such as a convoluted *DELETE* operation or a very complex *UPDATE*, can be solved by using clever *WHERE* and *JOIN* clauses instead of cursors.

DB2

DB2 cursors are implicitly forward-only cursors that always scroll forward one record at a time. A DB2 cursor, when compared to the ANSI standard, is essentially a *FETCH NEXT 1* cursor. DB2 does not support keywords like *PRIOR*, *ABSOLUTE*, and *RELATIVE*. DB2 *FETCH* syntax is:

```
FETCH [ FROM ] cursor_name  
  
{ [INTO variable1 [, ...] ] |  
  
  [USING DESCRIPTOR descriptor_name] }
```

where:

USING DESCRIPTOR *descriptor_name*

Identifies a previously created SQLDA (SQL Descriptor Area) called *descriptor_name* to contain zero or more of the retrieved values from the *FETCH* statement.

When saving fetched values *INTO* variables, DB2 allows *LOB* values to be inserted into regular host variables such as

VARCHAR (when they are big enough to hold the value), locator variables, or file-reference variables.

When saving fetched values with the *USING DESCRIPTOR* clause, you should remember several rules about descriptor areas:

- *SQLN* indicates the number of *SQLVAR* occurrences in the *SQLDA*.
- *SQLDABC* indicates the number of bytes of storage allocated in the *SQLDA*.
- *SQLD* indicates the number of variables used in the *SQLDA* when processing the statement. *SQLD* must be set to a value greater than or equal to zero and less than or equal to *SQLN*.
- *SQLVAR* indicates the attributes of the variables. The descriptor needs enough storage to hold all *SQLVAR* occurrences. Use this formula to calculate the size of *SQLDABC*: it must be greater than or equal to $16 + \text{SQLN} * (n)$, where n is the length of *SQLVAR* occurrences.
- When *LOB* or structured type result columns must be accommodated, use two *SQLVAR* entries for every select item (or column returned from the result table).

For example, the following *FETCH* statement uses an *SQLDA*:

```
FETCH employee_cursor USING DESCRIPTOR :sqlda2
```

In the above example, the descriptor **:sqlda2** and the cursor **employee_cursor** must already exist.

MySQL

Not supported.

Oracle

Oracle cursors are implicitly forward-only cursors that always scroll forward one record at a time. An Oracle cursor, when compared to the ANSI standard, is essentially a *FETCH NEXT 1* cursor. Oracle cursors must either insert the retrieved values into matching variables, or use the *BULK COLLECT* clause to insert all of the records of the result set into an array. Oracle does not support keywords like *PRIOR*, *ABSOLUTE*, and *RELATIVE*. However, Oracle does support both forward-only and scrollable cursors in the database via the Oracle Call Interface (OCI). OCI also supports features like *PRIOR*, *ABSOLUTE*, and *RELATIVE* for read-only cursors whose result set is based on a read consistent snapshot.

Oracle's *FETCH* syntax is:

```
FETCH cursor_name
```

```
{ INTO variable_name1 [...] | BULK COLLECT INTO
```

```
  [collection_name [...] [LIMIT int] }
```

where:

```
BULK COLLECT INTO collection_name
```

Retrieves the entire rowset, or a specified number of rows (see *LIMIT*), into a client-side array or collection variable named *collection_name*.

```
LIMIT int
```

Limits the number of records fetched from the database when using the *BULK* statement. The *int* value is a nonzero integer (or a variable representing an integer value).

Oracle supports dynamic SQL-based cursors whose text can be built at runtime. Oracle also supports *cursor_names* that may be any allowable variable, parameter, or host-array type. In fact, you may also use user-defined or *%ROWTYPE* record variables for the *INTO* clause as well. This allows you to construct flexible, dynamic cursors in a template-like structure.

For example:

```
DECLARE

    TYPE namelist IS TABLE OF employee.lname%TYPE;

    names namelist;

    CURSOR employee_cursor IS SELECT lname FROM employee;

BEGIN

    OPEN employee_cursor;

    FETCH employee_cursor BULK COLLECT INTO names;

    ...

    CLOSE employee_cursor;

END;
```

Oracle *FETCH* often is paired with a PL/SQL *FOR* loop (or other kind of loop) to cycle through all the rows in the cursor. You should use the cursor attributes *%FOUND* or *%NOTFOUND* to detect the end of the rowset. For example:

```
DECLARE

    TYPE employee_cursor IS REF CURSOR RETURN employee%ROWTYPE;

    employee_cursor EmpCurTyp;

    employee_rec    employee%ROWTYPE;

BEGIN

    LOOP

        FETCH employee_cursor INTO employee_rec;

        EXIT WHEN employee_cursor%NOTFOUND;

        ...

    END LOOP;

CLOSE employee_cursor;

END;
```

This example uses a standard PL/SQL loop with an *EXIT* clause to end the loop when there are no more rows in the cursor to process.

PostgreSQL

PostgreSQL supports both forward and backward scrolling cursors. It covers the ANSI standard except *ABSOLUTE*, *FIRST*, or *LAST*. The syntax for *FETCH* in PostgreSQL is:

```
FETCH { [ FORWARD | BACKWARD | ABSOLUTE | RELATIVE ] }

    { [int | ALL | NEXT | PRIOR ] }

{ IN | FROM } cursor_name

[ INTO :variable1 [,...] ]
```

where:

FORWARD

Tells PostgreSQL that the cursor scrolls forward. This is the default if no other mode is defined.

BACKWARD

Tells PostgreSQL that the cursor scrolls backward.

ABSOLUTE

Allowed as a keyword, but not supported in functionality.

RELATIVE

Allowed as a keyword, but not supported in functionality.

int

A signed integer that indicates how many records to scroll forward (if positive) or backward (if negative).

ALL

Retrieves all records remaining in the cursor.

NEXT

Retrieves the next single record.

PRIOR

Retrieves the previous single record.

IN | FROM cursor_name

Defines the previously declared and opened cursor where data is retrieved.

INTO variable

Assigns a cursor value to a specific variable. As with ANSI cursors, the values retrieved by the cursor and the variables must match in number and datatype.

PostgreSQL cursors must be used with transactions explicitly declared using *BEGIN*, and must be closed with *COMMIT* or *ROLLBACK*.

The following PostgreSQL statement retrieves five records from the **employee** table and displays them:

```
FETCH FORWARD 5 IN employee_cursor;
```

It is important to note that PostgreSQL also supports a separate command, *MOVE*, to move to a specific cursor position. It differs from *FETCH* only by not returning values into variables:

```
MOVE { [ FORWARD | BACKWARD | ABSOLUTE | RELATIVE ] }
```

```
    { [ int | ALL | NEXT | PRIOR ] }
```

```
{ IN | FROM } cursor_name
```

For example, the following code declares a cursor, skips forward five records, and then returns the values in the sixth record:

```
BEGIN WORK;

  DECLARE employee_cursor CURSOR FOR SELECT * FROM employee;

  MOVE FORWARD 5 IN employee_cursor;

  FETCH 1 IN employee_cursor;

  CLOSE employee_cursor;

COMMIT WORK;
```

The above code will return a single row from the **employee_cursor** result set.

SQL Server

SQL Server supports a variation of *FETCH* that is very close to the ANSI standard:

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |
        { ABSOLUTE | RELATIVE int } }
      [FROM] [GLOBAL] cursor_name
      [INTO @variable1 [, ...] ]
```

The differences between SQL Server's implementation and the ANSI standard are very small. First, SQL Server allows you to use variables in place of the *int* and *cursor_name* values. Second, SQL Server allows the declaration and use of *GLOBAL* cursors that can be accessed by any user or session, not just the one that created it.

There are some rules that apply to how you use the *FETCH* command based upon how you issued the *DECLARE CURSOR* command:

- When you declare a *SCROLL* SQL-92 cursor, all *FETCH* options are supported. In all other SQL-92 cursors, only *NEXT* is the only option supported. (There is also an alternate Transact-SQL style *DECLARE CURSOR* statement.)
- *DYNAMIC SCROLL* cursors support all *FETCH* options except *ABSOLUTE*.
- *FORWARD-ONLY* and *FAST_FORWARD* only support *FETCH NEXT*.
- *KEYSET*, *STATIC*, and *SCROLL* cursors (and *DYNAMIC*, *FORWARD-ONLY*, and *FAST_FORWARD* are not defined), all *FETCH* options are supported.



SQL Server also requires a *DEALLOCATE* statement, in addition to *CLOSE*, to release memory consumed by a cursor.

Here's a full example that goes from declaring and opening a cursor to initiating several fetches and then finally closing and deallocating the cursor:

```
DECLARE @vc_lname VARCHAR(30), @vc_fname VARCHAR(30), @i_emp_id CHAR(5)

DECLARE employee_cursor SCROLL CURSOR FOR

  SELECT lname, fname, emp_id

  FROM employee

  WHERE hire_date <= 'FEB-14-2004'

OPEN employee_cursor

-- Fetch the last row in the cursor.
```

```
FETCH LAST FROM employee_cursor
-- Fetch the row immediately prior to the current row in the cursor.
FETCH PRIOR FROM employee_cursor
-- Fetch the fifth row in the cursor.
FETCH ABSOLUTE 5 FROM employee_cursor
-- Fetch the row that is two rows after the current row.
FETCH RELATIVE 2 FROM employee_cursor
-- Fetch values eight rows prior to the current row into variables.
FETCH RELATIVE -8 FROM employee_cursor
INTO @vc_lname, @vc_fname, @i_emp_id
CLOSE employee_cursor
DEALLOCATE employee_cursor
GO
```

Remember that in SQL Server you must not only *CLOSE* the cursor, but you must also *DEALLOCATE* it. In some rare cases, you might wish to reopen a closed cursor. You can reuse any cursor that you have closed but not deallocated. The cursor is permanently destroyed only when it is deallocated.

See Also

```
CLOSE
DECLARE CURSOR
OPEN
```

GRANT Statement

The *GRANT* statement assigns privileges to users and roles, which allows them to access and use database objects (that is, object privileges).

In addition, most database platforms use the *GRANT* statement to authorize users and roles to create database objects and execute stored procedures, functions, and so on (particularly usage privileges).

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 syntax for *GRANT* includes only the assignment of object privileges and roles to a specific user:

```
GRANT { {object_privilege [,...]} | role [,...]} }
```

```
[ON database_object_name]
```

```
[TO grantee [,...]]
```

[WITH HIERARCHY OPTION] [WITH GRANT OPTION] [WITH ADMIN OPTION]

[FROM { CURRENT_USER | CURRENT_ROLE }]

Keywords

object privilege

Grants one or more privileges for a SQL statement. Multiple privileges may be granted at one time, as long as commas separate them. Do not combine *ALL PRIVILEGES* with other privileges, but the remainder may be combined in any order. Valid privileges are:

ALL PRIVILEGES

Indicates a shorthand for every privilege the grantor has available to grant. Grants all privileges currently assigned to the grantor to the named users and/or for the specified database objects. This is generally not a recommended approach, since it can encourage sloppy permissioning.

EXECUTE

Grants the privileges to execute any ANSI SQL *routine*. An ANSI routine is any stored procedure, user-defined function, or user-defined method.

{SELECT | INSERT | UPDATE | DELETE}

Grants the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns of a given table on which the privilege will be granted (except for the *DELETE* statement).

REFERENCES

Grants the privilege to use the column(s) in any constraint or assertion, not just a foreign key. The *REFERENCES* clause also allows you to define column-specific details (by including a parenthetical, comma-delimited list) for statements like *INSERT*, *SELECT*, and so on. Finally, *REFERENCES* grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

TRIGGER

Grants the privilege of creating a trigger to operate on a specific table and its columns.

UNDER

Grants the privilege to create subtypes or typed tables.

USAGE

Grants privileges to use a domain, user-defined type, character set, collation, sequence, or translation.

role

Grants a specific, predefined role from the named grantee identified in the *FROM* clause. For example, the database administrator might want to create a role called **Reporter** that has read-only access to several tables. By granting this role to a user, that user then receives all of the read-only permissions granted to the **Reporter** role. Users can have a role assigned to them and receive all of the permissions already assigned to that role.

ON *database_object_name*

Grants privileges on the specified, predefined database object identified by *database_object_name*, which may be

one of the following:

```
{[TABLE] object_name | DOMAIN object_name |  
COLLATION object_name | CHARACTER SET object_name |  
TRANSLATION object_name | SPECIFIC ROUTINE routine_name}
```

TO *grantee*

Assigns the privilege to the named user(s) or role(s), designated as *grantee*. You can assign privileges to multiple users and/or roles, as long as commas separate them. Alternately, privileges may be granted to *PUBLIC*, meaning that all users (including those that will be created in the future) have the specified privileges.

WITH HIERARCHY OPTION

Bestows the *WITH HIERARCHY OPTION* privilege, enabling the grantee to *SELECT* not only from the named table, but also from all of its subtables. This applies to privilege grants only.

WITH GRANT OPTION

Enables the grantee to further grant the privileges to other users. This applies to privilege grants only.

WITH ADMIN OPTION

Bestows the ability to assign a role. The ability to assign role responsibilities is separate from the ability to grant privileges to other users or roles.

FROM { *CURRENT_USER* | *CURRENT_ROLE*}

Names the user who is granting the specified privilege(s), either the *CURRENT_USER* or the *CURRENT_ROLE*. This clause is optional and assumes the current user context.

Rules at a Glance

You may grant many privileges on a single object with one statement, but do not mix the *ALL PRIVILEGES* privilege with the individual SQL statement keywords in a single *GRANT* statement. You can grant a specific privilege on a specific database object to a single user using:

```
GRANT privilege_name ON object_name TO grantee_name
```

For example:

```
GRANT SELECT ON employee TO Dylan;
```

You can grant a specific privilege on a specific object to all users via the *PUBLIC* user. When a grant is issued to *PUBLIC*, that means everyone has permission to the object without the need for a grant specifically to the object. For example:

```
GRANT SELECT ON employee TO PUBLIC;
```

When granting privileges to multiple grantees, simply place a comma between each:

```
GRANT SELECT ON employee TO Dylan, Matt
```

The previous example shows that you may grant privileges to multiple users, in this case **Dylan** and **Matt**, and to the *PUBLIC* user in a single *GRANT* statement. When granting privileges on a table, the privileges may extend to the column level by showing a list of columns enclosed in parentheses after the table name. For example:

```
GRANT SELECT ON employee(emp_id, emp_fname, emp_lname, job_id)
```

```
TO Dylan, Matt
```

The previous example shows that **Dylan** and **Matt** now have select privileges on several columns of the **employee** table.

Programming Tips and Gotchas

Depending on the specific database implementation, views may or may not have independent access privileges from their base tables.

Note that all of the optional *WITH* clauses control extensions to the base privilege(s) granted in the statement. Thus, the following command grants the privilege to *SELECT* records from the **employee** table:

```
GRANT SELECT ON employee TO Dylan;
```

but the following command grants the privilege to *SELECT* records from the **employee** table *and* also to grant that *SELECT* privilege to other users.

```
GRANT SELECT ON employee TO Dylan
```

```
WITH GRANT OPTION;
```

Similarly, you can use the *REVOKE* statement to revoke only the *WITH GRANT OPTION*, the individual *SELECT* privilege, or both.

Most of the platforms isolate privileges at the user and role level. Thus, an individual user who is a member of two roles may have an individual set of permissions four times — once for themselves, once for *PUBLIC*, and once with each of the roles for which they have permission. In this situation, you should be very careful. In order to remove a specific privilege completely from such a user, you have to revoke the user's privileges directly, and then in both roles.

DB2

DB2 offers many options in its implementation of the *GRANT* command. It is important to note that privileges on database objects (such as *SELECT* privileges on a specific table) are handled by the command, as are a variety of other privileges:

```
GRANT { [ database_privilege [,...] ON DATABASE ] |  
  [ CONTROL ON INDEX index_name ] |  
  [ {BIND | CONTROL | EXECUTE} ON PACKAGE package_name ] |  
  [ EXECUTE ON [SPECIFIC] {FUNCTION | METHOD {* | FOR} | PROCEDURE}  
  object_name ] |  
  [ {ALTERIN | CREATEIN | DROPIN} ON SCHEMA schema_name ] |  
  [ ON SEQUENCE sequence_name ] |  
  [ PASSTHRU ON SERVER server_name ] |  
  [ object_privilege [,...] ON [TABLE] object_name ] }  
TO [ {USER |GROUP} ]  
  { grantee_name [,...] | group_name [,...] | PUBLIC } ]  
[WITH GRANT OPTION] | [ USE OF TABLESPACE tablespace_name ]
```

The parameters are as follows:

database_privileges

Grants one or more privileges that apply to the entire database, rather than to a single object. The database privileges are:

BINDADD

Grants the authority to create packages. The user who creates a package always has the *CONTROL* privilege on that package even when *BINDADD* is revoked at a later date.

CONNECT

Grants the authority to access the specified database.

CREATETAB

Grants the authority to create a base table. The user who created the base table always has the *CONTROL* privilege on the table even if *CREATETAB* is revoked at a later date.

CREATE_EXTERNAL_ROUTINE

Grants the privilege to register external routines. This is somewhat dangerous since external threads can have adverse side effects. The routine will continue to exist once created, even if the *CREATE_EXTERNAL_ROUTINE* privilege is revoked for the user. (For more information, check on the *THREADSAFE* clause of the *CREATE* or *ALTER* routine statements in the DB2 documentation.)

CREATE_NOT_FENCED_ROUTINE

Grants the authority to create routines that are not fenced into a specific user process. That is, the routines may execute within the database manager's process. *CREATE_NOT_FENCED_ROUTINE* is a superset of *CREATE_EXTERNAL_ROUTINE*. Consequently, you automatically are granted the latter privilege when assigned the former. Read the description of *CREATE_EXTERNAL_ROUTINE* for more information.

DBADM

Grants superuser authority, including the ability to create any object, and including all privileges against all objects in the database, even the ability to grant those privileges to other users.

IMPLICIT_SCHEMA

Grants the authority to create a schema implicitly, though this privilege does not impact the ability to create a schema or any objects within a schema explicitly.

LOAD

Grants the authority to explicitly *LOAD* the specified database. *SYSADM* and *DBADM*, who automatically possess this privilege, need no other explicitly granted privileges to load a database. But other users with *LOAD* privilege must have, at a minimum, *INSERT* privileges on tables they wish to load.

QUIESCE_CONNECT

Grants the privilege to access the specified database while it is quiesced.

CONTROL ON INDEX *index_name*

Grants the privilege to drop an index, once created, of the specified *index_name*. This is the explicit *CONTROL* privilege for indexes. It is automatically granted to the creator of any index.

{BIND | CONTROL | EXECUTE} ON PACKAGE *package_name*

Grants the privileges associated with packages. Note that *RUN* and *PROGRAM* are acceptable synonyms for *EXECUTE* and *PACKAGE*, respectively. The package privileges are:

BIND

Grants the authority to bind or rebind the specified package. Since any SQL statements in the package are checked at bind time, the user must also have statement permissions on each object referenced by the package.

CONTROL

Grants the authority to rebind, drop, or execute the specified package, as well as to grant *BIND* and *EXECUTE* privileges to other users. Users granted the *CONTROL* privilege automatically receive *BIND* and *EXECUTE* as well. *CONTROL* is automatically granted to the creator of a package.

EXECUTE

Grants the authority to execute the specified package.

EXECUTE ON [SPECIFIC] {FUNCTION | METHOD {* | FOR} | PROCEDURE} *object_name*

Grants privileges on an ANSI routine (i.e., user-defined function, user-defined method, or stored procedure). The elements of this clause are as follows:

FUNCTION [*schema.*]*function_name*

Specifies that the user is granted execution privileges on a specific function. A *SPECIFIC* function name may be specified. DB2 also supports functions without names. In this case, *FUNCTION [(param [...])]* should be used, including any and all parameters of the function. If *schema* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific function name to grant a user the execution privilege on all functions in the current or specified schema.

METHOD { * | FOR *type_name*}

Specifies that the user is granted rights to execute all methods of *type_name*, even those that have not yet been created. The optional *FOR type_name* clause names the type where the specified method is found. A *SPECIFIC* method name may be specified. DB2 also supports methods without names. In this case, *METHOD [(param [...])]* should be used, including any and all parameters of the function. Alternately, the asterisk (*) can be used to specify that all types in the schema are granted to the user.

PROCEDURE [*schema.*]*procedure_name*

Specifies that the user be granted execute privileges on all stored procedures in the specified schema, even those created in the future. If *schema* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific stored procedure name to grant a user the execution privilege on all stored procedures in the current or specified schema.

{ALTERIN | CREATEIN | DROPIN} ON SCHEMA *schema_name*

Grants privileges associated with manipulating a schema (that is, a user-owned set of objects). Only users with *SYSADM*, *DBADM*, or *WITH GRANT OPTION* permissions for all privileges for the specified schema may issue this command. The elements are as follows:

ALTERIN

Grants the authority to alter or comment on objects in a schema defined by *ON SCHEMA schema_name*.

CREATEIN

Grants the authority to create objects in a schema defined by *ON SCHEMA schema_name*.

DROPIN

Grants the authority to drop objects in the schema defined by *ON SCHEMA schema_name*.

{ALTER | USAGE} ON SEQUENCE *sequence_name* TO PUBLIC

Grants privileges on a sequence in an explicit or current schema (*USAGE*) or the privilege to execute the *ALTER SEQUENCE* statement (*ALTER*).

PASSTHRU ON SERVER *server_name*

Grants the privilege to access an external data source named *server_name* using pass-through mode.

object_privilege

Represents privileges for a variety of statements on tables, views, and nicknamed tables or views. Object privileges may be combined in any order:

ALL [PRIVILEGES]

Provides a shorthand for every privilege the grantor has available to grant. The *PRIVILEGES* keyword is optional. It grants all privileges currently assigned to the granting user for the specified database objects, including *SELECT*, *INSERT*, *UPDATE*, and *DELETE* privileges. With *ALL*, you have privileges to add columns to a table, create or drop primary keys or unique constraints, create or drop foreign keys, create or drop check constraints, create triggers on a table, add or change comments on a table, change a nicknamed column name or datatype, or add, rest, or drop a nicknamed column name. However, granting *ALL* is generally not a good approach since it can encourage sloppy coding.

ALTER

The privilege to add or alter columns, constraints, comments, or triggers on a specified table or view. *ALTER* also permits the use of nicknames on column names or datatypes or to change a column option for a nickname.

CONTROL

Grants a large number of privileges to the specific user or role (or *PUBLIC*) for the specified object. When you grant *CONTROL* on a table, that includes the privileges for *ALTER*, *DELETE*, *INSERT*, *INDEX*, *REFERENCES*, *SELECT*, and *UPDATE*. When you grant *CONTROL* on a view, that includes *DELETE*, *INSERT*, *SELECT*, and *UPDATE*. On nicknames, *CONTROL* includes *ALTER*, *INDEX*, and *REFERENCES*.

The grantee has the ability to grant these aforementioned privileges to others, who may drop the specified table, view, or nickname. They may execute the *RUNSTATS* utility on tables and indexes, as well as execute the *SET INTEGRITY* command on tables, materialized views, and staging tables.

INDEX

The ability to create an index against a table, or to create an index specification on a nickname. (The creator of an index retains *CONTROL* privileges on the indexes they create, even if the *INDEX* privilege is later revoked.)

{*SELECT* | *INSERT* | *UPDATE* | *DELETE*}

Grants the specific privilege to a given user for the specified database object, such as a table or view. In some cases, these privileges also extend to utilities with similar behavior. For example, the *INDEX* privilege also includes the use of the *IMPORT* utility. You may add a parenthetical list of the columns on a given table where the privilege is granted. The *UPDATE* privilege may be assigned down to the column level by including a parentheses-encapsulated list of column or columns made available to the user.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object. The *REFERENCES* privilege may be assigned down to the column level by including a parentheses-encapsulated list of column or columns made available to the grantee.

TO { [USER | GROUP]} *grantee_name*

Names the user or users who will gain the specified privilege or privileges. When both a user and group exist with the same name, use the optional *USER* or *GROUP* keywords to control whether the privileges are granted to the user or to the group. *PUBLIC* may also be used to grant privileges to the *PUBLIC* role. Multiple grantees can be listed; separate them with a comma between each.

WITH GRANT OPTION

Entitles the grantee to assign the granted privilege(s) to other users.

USE OF TABLESPACE *tablespace_name*

Grants the privilege of specifying or defaulting new tables to the specified *tablespace_name*. In effect, it enables the *USE* privilege on that tablespace.

In situations when a group and a user of the same name exist, you must use either *GROUP* or *KEYWORD* to inform DB2 whether you want the privileges assigned to the group or to the user. If you do not, DB2 will return an error and abort the *GRANT* statement. For example, to grant *CONNECT* and *BINDADD* permissions to the user **emily** and the group **datareaders**:

```
GRANT CONNECT, BINDADD ON DATABASE TO USER emily,  
GROUP datareaders
```

To give the user **Dylan**, and all users who are members of the **Admin** role, control over a specific index:

```
GRANT CONTROL ON INDEX emp_ndx TO dylan, admin;
```

You can give a user rights on a package to execute it and bind (or rebind it), without the privilege to drop it (via *CONTROL*) as follows:

```
GRANT BIND, EXECUTE ON PACKAGE d02.sales_processing  
TO USER dylan
```

The following example grants a user a variety of privileges on the **sales** table, as well as the privilege to pass those privileges on to other users:

```
GRANT DELETE, INSERT, SELECT, UPDATE ON sales TO USER sam  
WITH GRANT OPTION
```

Similarly, you can give another user privileges on a specific object (a view in this case) that includes only specific columns in the object:

```
GRANT UPDATE (sales_qty, sales_discount) ON sales TO USER jake
```



DB2 also has privileges at the instance level that aren't managed with the *GRANT* statement. Most importantly, there are three database manager configuration parameters (*SYSADM_GRP*, *SYSCTRL_GRP*, and *SYSMAINT_GRP*) that control administrative rights to the instance. You control these privileges using the DB2 *UPDATE DATABASE MANAGER CONFIGURATION* statement. For example:

```
ATTACH DB2;
```

```
UPDATE DATABASE MANAGER CONFIGURATION
USING
  sysadm_group 'DB2GRP'
IMMEDIATE;

DETACH;
```

MySQL

MySQL provides additional access privileges, primarily relating to object manipulation within a database. *GRANT* in MySQL is available from Version 3.22.11 and on. The syntax is as follows:

```
GRANT [ { ALL [PRIVILEGES] |
  {SELECT | INSERT | UPDATE} [ (column_name [...]) ] | DELETE |
  REFERENCES [ (column_name [...]) ] } |
[ USAGE ] | [{ALTER | CREATE | DROP}] | [FILE] | [INDEX] | [PROCESS] |
[RELOAD] | [SHUTDOWN] | [CREATE TEMPORARY TABLES] | [LOCK TABLES] |
[REPLICATION CLIENT] | [REPLICATION SLAVE] | [SHOW DATABASES] |
[SUPER] },...]
ON {table_name | * | *.* | database_name.*}
[TO grantee_name [IDENTIFIED BY 'password']] [,...]
[WITH GRANT OPTION]
```

where:

ALL [PRIVILEGES]

Grants a user all applicable privileges that apply at the current level on one or more objects in the database, with the exception of *WITH GRANT OPTION*. For example, *GRANT ALL ON *.** grants all privileges, except *WITH GRANT OPTION*, globally.

SELECT | INSERT | UPDATE | DELETE

Grants the ability to read, write, modify, and erase data from tables (and possibly from specific columns of a table), respectively. Column-level permissions are allowed on *SELECT*, *INSERT*, and *UPDATE*, but not *DELETE*. Granting column-level permissions can slow down MySQL somewhat, since more permission checking is required.

REFERENCES

Remains unimplemented.

USAGE

Creates a "no privilege" user account for use in creating a user that has no privileges.

ALTER | CREATE | DROP

Grants the ability to alter, drop, or create tables and other database objects. When using the *CREATE* syntax, you are not required to specify an *ON* clause to identify a specific table.

FILE

Grants the ability to load data from, or write data to, files using the *SELECT INTO* and *LOAD DATA* commands.

INDEX

Grants the ability to create or drop indexes.

PROCESS

Grants the ability to view running processes using *SHOW FULL PROCESSLIST*.

RELOAD

Grants the ability to invoke the *FLUSH* and *RESET* commands.

SHUTDOWN

Grants usage of the *mysqladmin shutdown* command to kill the server process.

CREATE TEMPORARY TABLES

Grants the ability to create temporary tables.

LOCK TABLES

Grants the use of the MySQL command *LOCK TABLES* on tables where the user has *SELECT* privileges.

REPLICATION CLIENT

Grants a user the privilege to see metadata about replication slaves and masters.

REPLICATION SLAVE

Grants a replication slave the right to read binlogs from a replication master.

SHOW DATABASES

Grants the user the privilege to execute the MySQL command *SHOW DATABASES*.

SUPER

Grants the user the privilege of one connection even if the *max_connections* threshold is reached. Users with *SUPER* privileges can also execute important MySQL commands like *CHANGE MASTER*, *KILL*, *mysqladmin debug*, *PURGE [MASTER] LOGS*, and *SET GLOBAL*.

*ON {table_name | * | *.* | database_name.*}*

Grants privileges from the specified *table_name*, from all tables within the current database using an asterisk (*), from all tables in all databases with **.**, or from all tables in the specified *database_name* with *database_name.**.

TO *grantee_name* [IDENTIFIED BY '*password*'] [,...]

Names the user or users who will gain the specified privilege. You may enter multiple users separated by commas. When granting privileges to a new user, you may also optionally set the new password for the new named user at the same time using the *IDENTIFIED BY* clause.

WITH GRANT OPTION

Entitles the grantee to assign the granted privilege to other users.

Since MySQL is focused on speed, you can implement server-wide features that provide high-speed performance. For example, you can enable the *SKIP_GRANT_TABLES* startup option to disable checks for privileges. This can speed up queries, but obviously no permission checking is done. This means that all users have full access to all resources in the database.

The following are access privileges that are usable with tables: *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE*, *DROP*, *GRANT*, *INDEX*, and *ALTER*. At the column level, you can apply *INSERT*, *UPDATE*, and *SELECT*.

In any situation, the host, table, database, and column names may each be up to 60 characters in size. The username (a.k.a. the *grantee_name*) may be up to 16 characters.

MySQL also supports the possibility of granting rights to a specific user on a specific host if the *grantee_name* is in the form *USER@HOST*. Wildcards can be included in the hostname of a *grantee_name* to provide the access privilege to a large number of users at one time. Also, a missing hostname is the same as the "%" wildcard. For example:

```
GRANT SELECT ON employee TO katie@% IDENTIFIED BY 'audi',  
    annalynn IDENTIFIED BY 'cadillac',  
    cody@us% IDENTIFIED BY 'bmw';
```

The preceding *GRANT* statement grants read privileges on the employee table to the user **katie** explicitly at any hostname. Similarly, the *GRANT* statements grants read privileges to the user **annalynn** implicitly at any hostname. And finally, the *GRANT* statement grants read privileges to the user **cody** explicitly at any hostname starting with "US".

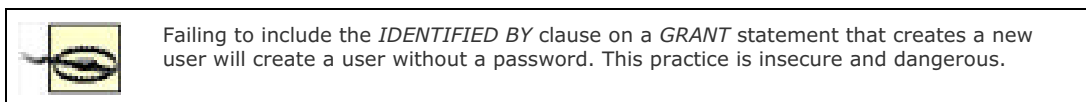
The *grantee_name* must be 16 or fewer characters. When specifying the user, password protection may be enforced by including the *IDENTIFIED BY* clause.

The following example grants permissions to three users with passwords:

```
GRANT SELECT ON employee TO dylan IDENTIFIED BY 'porsche',  
    kelly IDENTIFIED BY 'mercedes',  
    emily IDENTIFIED BY 'saab';
```

In MySQL, if you grant permissions to a user that doesn't exist, it creates that user. The two statements that follow create a user and password with global privileges and another user that has no privileges (the *USAGE* privilege):

```
GRANT SELECT ON *.* TO tony IDENTIFIED BY 'humbolt';  
GRANT USAGE ON sales.* TO alicia IDENTIFIED BY 'dakota';
```



If you do create a new user using the *GRANT* command, and you forget to specify a password, you can, and probably should, set a password using the *SET PASSWORD* command.

MySQL also allows you to issue grants on remote hosts, specified with *@hostname*, and include the wildcard character percent (%). For example, the following grant provides all permissions on all tables of the current database across all hosts in the **notforprofit.org** domain to the user **annalynn**:

```
GRANT ALL ON * TO annalynn@%.notforprofit.org;
```

Privileges are written directly to system tables at four different levels:

Global level

Global privileges apply to all databases on a given server and are stored in the **mysql.user** system table.

Database level

Database privileges apply to all tables in a given database and are stored in the **mysql.db** system table.

Table level

Table privileges apply to all columns in a given table and are stored in the **mysql.tables_priv** system table.

Column level

Column privileges apply to single columns in a given table and are stored in the **mysql.columns_priv** system table.

Because of the usage and system tables involved, you may grant redundant permissions at different levels. For example, you could grant a user *SELECT* privileges on a specific table, and then grant global *SELECT* privileges on that same table to all users of the database. You would then have to revoke those privileges separately to ensure the user no longer had the privilege of *SELECT*ing from the table.

The system tables are also somewhat unprotected, so it is possible to change permissions by issuing *INSERT*, *UPDATE*, and *DELETE* statements against the tables rather than by using *GRANT* or *REVOKE*. Normally, all privileges are read into memory when MySQL starts. Database, table, and column privileges issued by a *GRANT* statement are available immediately and take effect right away. User privileges issued by a *GRANT* statement are noticed immediately and take effect the next time the user connects. If you directly modify permissions in the system tables, none of the changes are noticed until you restart MySQL, or until you issue a *FLUSH PRIVILEGES* command.

Oracle

Oracle's implementation of the *GRANT* statement supports an enormous number of variations and permutations. The syntax is as follows:

```
GRANT { [object_privilege] [,...] |  
       [system_privilege] [,...] |  
       [role] [,...] }  
[ ON { [schema_name.][object] [,...] |  
       [DIRECTORY directory_object_name] |  
       [JAVA { SOURCE | RESOURCE } ] [schema_name.][object] }  
TO { {grantee_name [,...] | role_name [,...] | PUBLIC}  
    [WITH { GRANT | HIERARCHY } OPTION]  
[IDENTIFIED BY password][WITH ADMIN OPTION];
```

You can grant multiple privileges with a single invocation of the command, but they must all be of the same type (object, system, or role).

For example, you could grant a user three object privileges on a single table in one *GRANT* statement, and then issue a separate grant of two system privileges to a role, and yet a third grant of several roles to a user, but not all of them in one *GRANT* statement.

Following are the parameters to Oracle's *GRANT* statement:

object_privilege

Grants the specified privileges to the specified *grantee_name* or role on the named schema object (for example, a table or view). You may combine several object privileges, schema objects, or grantees in a single statement. However, you may *not* grant system privileges or roles in the same statement as object privileges. The object privileges are:

ALL [PRIVILEGES]

Grants all available privileges on a named schema object and is available for use on tables.

ALTER

Grants the privilege to change an existing table using the *ALTER TABLE* statement, and is available for use on tables and sequences.

DEBUG

Grants access to a table through a debugger. The access granted applies to any triggers on the table or to information on SQL that directly references the table and is available for use on tables, views, procedures, functions, packages, Java objects, and types.

EXECUTE

Grants the privilege to execute a stored procedure, user-defined function, or a package, and is available for use on procedures, functions, packages, Java objects, libraries, types, indextypes, and user-defined operators.

INDEX

Grants the privilege to create indexes on a table.

{ ON COMMIT REFRESH | QUERY REWRITE }

Grants the privilege to create a refresh-on-commit materialized view or to create a materialized view for query rewrite on the specified table. It is available for materialized views only.

{ READ | WRITE }

Grants the privilege to read and write files against a named directory including full path name of the operating system directory. Since Oracle has the ability to store files outside of the database, the Oracle server process must run under a user context with privileges on the applicable directories. You can enable Oracle to enforce security on individual users via this security mechanism. Note that *WRITE* is only useful in connection with an external table, such as a logfile or an error file.

REFERENCES

Grants the privilege to define referential integrity constraints, and is available for use on tables.

{SELECT | INSERT | DELETE | UPDATE}

Grants the privilege to execute each of the respective SQL commands against the named schema object and is available for use on tables, views, sequences (*SELECT* only), and materialized views (*SELECT* only). Note that you must grant *SELECT* privileges to a user or role that requires *DELETE* privileges. You can grant column-level privileges, as a parentheses-encapsulated list following the object name, only when granting *INSERT*, *REFERENCES*, or *UPDATE* object privileges on a table or view.

UNDER

Grants the privilege to create a subview under the named view and is available for use on views and types only.

Grants the specified Oracle system privilege to one or more users or roles. For example, you could grant the *CREATE TRIGGER* privilege or the *ALTER USER* privilege. In each case, issuing a grant for a system privilege empowers the user or role to execute the command of the same name. Refer to [Table 3-2](#) later in this section for a full list of system privileges.

role

Grants a role to the user or to another role. In addition to user-defined roles, there are a number of system roles that come predefined with Oracle:

CONNECT, RESOURCE, and DBA

Provided for backward compatibility with older versions of Oracle. Do not use these roles in current or newer versions of Oracle since they may be dropped in the future.

DELETE_CATALOG_ROLE, EXECUTE_CATALOG_ROLE, and SELECT_CATALOG_ROLE

Grant members of these roles the privilege deleting, executing, and selecting from data dictionary views and packages.

EXP_FULL_DATABASE and IMP_FULL_DATABASE

Grant members of these roles the privilege of executing the import and export utilities.

AQ_USER_ROLE and AC_ADMINISTRATOR_ROLE

Grant members of these roles the privilege to use or administrate Oracle's Advanced Queuing functionality.

SNMPAGENT

Assigned only to Oracle Enterprise Manager and Intelligent Agent.

RECOVERY_CATALOG_OWNER

Grants the privilege to create users who own their own recovery catalog.

HS_ADMIN_ROLE

Grants the privilege to access areas of the data dictionary used to support Oracle's heterogeneous services feature

ON *schema_name*

Grants the named privilege to a user or role on an object in the schema. Database objects include tables, views, sequences, stored procedures, user-defined functions, packages, materialized views, user-defined types, libraries, indextypes, user-defined operators, or a synonym for any of these. If you do not include the schema name, Oracle assumes the schema of the current user.

Oracle also supports two additional keywords for special cases:

DIRECTORY

Grants privileges on a directory object, which is an Oracle object corresponding to a filesystem directory.

JAVA

Grants privileges on a Java *SOURCE* or a *RESOURCE* schema object.

TO { *grantee_name* | *role_name* | PUBLIC } [WITH { GRANT | HIERARCHY } OPTION]

Names the user or role who will get the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role. Multiple grantees can be listed with a comma between each, as in the following:

WITH GRANT OPTION

Enables the grantee to grant the specific privilege(s) to other users or to *PUBLIC*, but not to roles.

WITH HIERARCHY OPTION

Enables the grantee to receive privileges on all sub-objects when granted privileges on a parent object. This includes sub-objects created in the future. You can use this option only when granting the *SELECT* object privilege.

IDENTIFIED BY *password* [WITH ADMIN OPTION]

Establishes or changes a password that the grantee must use to enable a *role* being granted.

WITH ADMIN OPTION

Enables the grantee to exercise control over a role that you are granting to that grantee. First, this clause enables the grantee to grant and revoke the role to and from other users or non-*GLOBAL* roles. It further allows the grantee to drop the role, or to change the authorization needed to access a role.

Grants to users take effect right away. Grants to roles take effect right away, assuming the roles are enabled. Otherwise, a grant to a role takes effect as the role is enabled. Note that you can grant roles to users and to other roles (including *PUBLIC*). For example:

```
GRANT sales_reader TO sales_manager;
```

To grant privileges on a view, you must have the specific privilege and the *WITH GRANT OPTION* on all of the base tables for the view.

Any time you want a privilege available to all users, simply grant it to *PUBLIC*:

```
GRANT SELECT ON work_schedule TO public;
```

However, there are some restrictions when granting system privileges and roles:

- A privilege or role should not appear more than once in the *GRANT* statement.
- Roles cannot be granted to themselves.
- Roles cannot be granted recursively. That is, you couldn't grant the role **sales_reader** to **sales_manager** and then grant **sales_manager** to **sales_reader**.

You can grant multiple privileges of the same type in a single statement. However, the grants must be on the same type of objects:

```
GRANT UPDATE (emp_id, job_id), REFERENCES (emp_id),  
ON employees  
TO sales_manager;
```

As an aside, granting any of the object permissions on a table allows the user (or role) to lock the table in any lock mode using Oracle's *LOCK TABLE* statement.

Nearly every supported Oracle feature or command is assignable with a *GRANT* command (as shown in [Table 3-2](#)). Privileges can be granted not only on database objects (such as tables and views) and system commands (such as *CREATE ANY TABLE*), but also on schema objects (such as *DIRECTORY*, *JAVA SOURCE*, and *RESOURCE*).

The *ANY* option, shown in [Table 3-2](#), grants the privilege to execute a given statement against objects of a specific type

owned by any user within the schema. Without the *ANY* option, the user can only execute the statement against objects in their own schema. A more complete list of Oracle system privileges is shown in [Table 3-2](#).

Table 3-2. Oracle system privileges

Options	Description
<i>CREATE CLUSTER</i>	Grants privilege to create a cluster in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY CLUSTER</i>	Grants privilege to create, alter, or drop a cluster in any schema, respectively.
{ <i>CREATE DROP</i> } <i>ANY CONTEXT</i>	Grants privilege to create or drop any context namespace, respectively.
<i>ALTER DATABASE</i>	Grants privilege to alter the database.
<i>ALTER SYSTEM</i>	Grants privilege to issue <i>ALTER SYSTEM</i> statements.
<i>AUDIT SYSTEM</i>	Grants privilege to issue <i>AUDIT sql_statements</i> statements.
<i>CREATE DATABASE LINK</i>	Grants privilege to create private database links in grantee's own schema.
<i>CREATE PUBLIC DATABASE LINK</i>	Grants privilege to create public database links.
<i>DROP PUBLIC DATABASE LINK</i>	Grants privilege to drop public database links.
<i>DEBUG CONNECT SESSION</i>	Grants privilege to connect the current session to the Java Debug Wire Protocol (JDWP) debugger.
<i>DEBUG ANY PROCEDURE</i>	Grants the privilege to debug all PL/SQL objects in the database, similar to granting <i>DEBUG</i> on all procedures, functions, and packages.
<i>CREATE DIMENSION</i>	Grants privilege to create dimensions in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY DIMENSION</i>	Grants privilege to create, alter, or drop dimensions in any schema.
{ <i>CREATE DROP</i> } <i>ANY DIRECTORY</i>	Grants privilege to create or drop directory database objects.
<i>CREATE ROLE</i>	Grants the privilege to create a database role.
<i>GRANT ANY ROLE</i>	Grants the privilege to grant a role to other users.
<i>CREATE INDEXTYPE</i>	Grants privilege to create an indextype in the grantee's own schema.
{ <i>CREATE DROP EXECUTE</i> } <i>ANY INDEXTYPE</i>	Grants privilege to create, alter, drop, or execute an indextype in any schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY INDEX</i>	Grants privilege to create, alter, or drop an index in any schema or an index on any table in any schema.
{ <i>CREATE DROP</i> } <i>ANY LIBRARY</i>	Grants privilege to create or drop external procedure/function libraries in any schema.
<i>CREATE MATERIALIZED VIEW</i>	Grants privilege to create a materialized view in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> }	

<i>ANY MATERIALIZED VIEW</i>	Grants privilege to create, alter, or drop materialized views in any schema.
<i>GLOBAL QUERY REWRITE</i>	Enables rewrite using a materialized view, or creates a function-based index, when materialized view or index reference tables or views are in any schema.
<i>QUERY REWRITE</i>	Enables rewrite using a materialized view, or creates a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
<i>ON COMMIT REFRESH</i>	Creates a refresh-on-commit materialized view on any table in the database or alters any refresh-on-demand materialized view into a refresh-on-commit materialized view.
<i>FLASHBACK ANY TABLE</i>	Grants the privilege to issue a SQL flashback query on any table, view, or materialized view in any schema. This privilege is not required for <i>DBMS_FLASHBACK</i> procedures.
<i>CREATE OPERATOR</i>	Grants privilege to create an operator and its bindings in the grantee's own schema.
{ <i>CREATE DROP EXECUTE</i> } <i>ANY OPERATOR</i>	Grants privilege to create, drop, or execute an operator and its bindings in any schema.
{ <i>CREATE DROP ALTER</i> } <i>ANY OUTLINE</i>	Grants privilege to create, alter, or drop outlines that can be used in any schema that uses outlines.
<i>SELECT ANY OUTLINE</i>	Grants the privilege to clone private outlines from public ones.
<i>CREATE PROCEDURE</i>	Grants privilege to create stored procedures, functions, and packages in grantee's own schema.
{ <i>CREATE ALTER DROP EXECUTE</i> } <i>ANY PROCEDURE</i>	Grants privilege to create, alter, drop, or execute stored procedures, functions, and packages in any schema.
{ <i>CREATE ALTER DROP</i> } <i>PROFILE</i>	Grants privilege to create, alter, or drop profiles.
<i>ALTER RESOURCE COST</i>	Sets costs for session resources.
{ <i>ALTER DROP</i> } <i>ANY ROLE</i>	Grants privilege to create, alter, drop, or grant roles in the database.
{ <i>CREATE ALTER DROP</i> } <i>ROLLBACK SEGMENT</i>	Grants privilege to create, alter, or drop rollback segments.
<i>CREATE SEQUENCE</i>	Grants privilege to create sequences in grantee's own schema.
{ <i>CREATE ALTER DROP SELECT</i> } <i>ANY SEQUENCE</i>	Grants privilege to create, alter, drop, or select sequence in any schema in the database.
<i>CREATE SESSION</i>	Grants privilege to connect to the database.
<i>ALTER SESSION</i>	Grants privilege to issue <i>ALTER SESSION</i> statements.
<i>RESTRICTED SESSION</i>	Logs on after the instance is started using the <i>SQL*Plus STARTUP RESTRICT</i> statement.
<i>CREATE SNAPSHOT</i>	Grants privilege to create snapshots in grantee's own schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY SNAPSHOT</i>	Grants privilege to create, alter, or drop snapshots (a.k.a. materialized views) in any schema.
<i>CREATE SYNONYM</i>	Grants privilege to create synonyms in grantee's own schema.

{CREATE DROP} ANY SYNONYM	Grants privilege to create or drop private synonyms in any schema.
{CREATE DROP} PUBLIC SYNONYM	Grants privilege to create or drop public synonyms.
CREATE ANY TABLE	Grants privilege to create tables in any schema. The owner of the schema containing the table must have a space quota on the tablespace to contain the table.
ALTER ANY TABLE	Grants privilege to alter any table or view in the schema.
BACKUP ANY TABLE	Uses the export utility to incrementally export objects from the schema of other users.
DELETE ANY TABLE	Deletes rows from tables, table partitions, or views in any schema.
DROP ANY TABLE	Grants privilege to drop or truncate tables or table partitions in any schema.
INSERT ANY TABLE	Grants privilege to insert rows into tables and views in any schema.
LOCK ANY TABLE	Grants privilege to lock tables and views in any schema.
UPDATE ANY TABLE	Grants privilege to update rows in tables and views in any schema.
SELECT ANY TABLE	Grants privilege to queries tables, views, or snapshots in any schema.
{CREATE ALTER DROP} TABLESPACE	Grants privilege to create, alter, or drop tablespaces.
MANAGE TABLESPACE	Grants privilege to take tablespaces offline and online, and begins and ends tablespace backups.
UNLIMITED TABLESPACE	Grants privilege to use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, the user's schema objects remain, but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
CREATE TRIGGER	Grants privilege to create a database trigger in grantee's own schema.
{CREATE DROP} ANY TRIGGER	Grants privilege to create or drop database triggers in any schema.
ALTER ANY TRIGGER	Grants privilege to enable, disable, or compile database triggers in any schema.
ADMINISTER DATABASE TRIGGER	Grants privilege to create a trigger on DATABASE. (You also must have the CREATE TRIGGER or CREATE ANY TRIGGER privilege.)
CREATE TYPE	Grants privilege to create object types and object-type bodies in grantee's own schema.
{CREATE ALTER DROP} ANY TYPE	Grants privilege to create, alter, or drop object types and object-type bodies in any schema.
EXECUTE ANY TYPE	Uses and references object types and collection types in any schema, and invokes methods of an object type in any schema <i>if you make the grant to a specific user</i> . If you grant EXECUTE ANY TYPE to a role, users holding the enabled role will <i>not</i> be able to invoke methods of an object type in any schema.
UNDER ANY TYPE	Creates subtypes under any parent object type, such as a table or view.
CREATE USER	Grants privilege to create users. This privilege also allows the creator to: <ul style="list-style-type: none"> • Assign quotas on any tablespace • Set default and temporary tablespaces • Assign a profile as part of a CREATE USER statement

<i>ALTER USER</i>	Grants the privilege to alter any user. This privilege authorizes the grantee to: <ul style="list-style-type: none"> • Change another user's password or authentication method • Assign quotas on <i>any</i> tablespace • Set default and temporary tablespaces • Assign a profile and default roles
<i>BECOME USER</i>	Grants privilege to become another user (required by any user performing a full database import).
<i>DROP USER</i>	Grants privilege to drop users.
<i>CREATE VIEW</i>	Grants privilege to create views in grantee's own schema.
<i>{CREATE DROP} ANY VIEW</i>	Grants privilege to create or drop views in any schema.
<i>UNDER ANY VIEW</i>	Creates subview under any parent view.
<i>ANALYZE ANY</i>	Analyzes any table, cluster, or index in any schema.
<i>AUDIT ANY</i>	Audits any object in any schema using <i>AUDIT schema_objects</i> statements.
<i>COMMENT ANY TABLE</i>	Allows comments on any table, view, or column in any schema.
<i>EXEMPT ACCESS POLICY</i>	Circumvents fine-grained access control. Generally, this is ill-advised since the user can then bypass any application-driven security policies.
<i>FORCE ANY TRANSACTION</i>	Forces the commit or rollback of any in-doubt distributed transaction in the local database and induces the failure of a distributed transaction.
<i>FORCE TRANSACTION</i>	Forces the commit or rollback of grantee's own in-doubt distributed transactions in the local database.
<i>GRANT ANY PRIVILEGE</i>	Grants any system privilege.
<i>GRANT ANY OBJECT PRIVILEGE</i>	Grants any object privilege available.
<i>RESUMABLE</i>	Enables resumable space allocations.
<i>SELECT ANY DICTIONARY</i>	Allows the user to query any data dictionary object in the SYS schema.
<i>SYSDBA</i>	Provides the user with <i>RESTRICTED SESSION</i> permission as well as authorizes the user to: <ul style="list-style-type: none"> • Perform <i>STARTUP</i> and <i>SHUTDOWN</i> operations • <i>ALTER DATABASE</i> • Open, mount, back up, or change character set • <i>CREATE DATABASE</i> • <i>ARCHIVELOG</i> and <i>RECOVERY</i>
<i>SYSOPER</i>	Provides the user with <i>RESTRICTED SESSION</i> permission as well as authorizes the user to: <ul style="list-style-type: none"> • Performs <i>STARTUP</i> and <i>SHUTDOWN</i> operations • <i>ALTER DATABASE, OPEN/MOUNT/BACKUP</i> • <i>ARCHIVELOG</i> and <i>RECOVERY</i>

Any of the privileges shown in [Table 3-2](#) containing the *ANY* keyword have special meaning. In effect, *ANY* gives a user the privilege to execute the specified command or operation in any schema. If you want to include all user schemas, but exclude the **SYS** system schema, keep the *O7_DICTIONARY_ACCESSIBILITY* initialization parameter at its default value of FALSE.

PostgreSQL

PostgreSQL supports a subset of the ANSI *GRANT* commands supporting object privileges against database objects such as tables. It does not support any of the *WITH* options available under the ANSI standard nor specialty features like *USAGE* and *UNDER*. PostgreSQL's *GRANT* syntax is as follows:

```
GRANT { ALL [PRIVILEGES]
| SELECT
| INSERT
| DELETE
| UPDATE
| RULE
| REFERENCES
| TRIGGERS
| USAGE } [,...]
ON [TABLE] { object_name } [,...]
TO {grantee_name | GROUP group_name | PUBLIC } [,...]
```

where:

ALL [PRIVILEGES]

Grants all privileges the grantor has available to grant. Using *ALL* is generally not a recommended approach, since it can encourage sloppy permissioning.

{SELECT | INSERT | DELETE | UPDATE}

Grants the specific privilege to a given user on the specified database object, such as a table or view. For column-level privileges, you may add a parenthetical list of the columns of a given table. The permission then applies only to the columns of the table that are named.

RULE

Grants the privilege to create or drop a rule on a table or view.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

TRIGGERS

Grants the privilege of creating a trigger to operate on a specific table and its columns.

USAGE

Grants privileges to use these objects: *CHARACTER SET*, *COLLATION*, *TRANSLATION*, and *DOMAIN* (user-defined datatype).

ON [TABLE] *object_name*

Specifies the object to which permissions are granted. The optional *TABLE* keyword is noise. You may grant permissions on more than one object at a time by specifying a comma-delimited list of object names.

TO {*grantee_name* | GROUP *group_name* | PUBLIC } [,...]

Names the user or role, or multiples thereof, that receive the specified privileges. In PostgreSQL, PUBLIC is a synonym for all users. PostgreSQL allows permissions to be assigned to a *GROUP* (essentially a role), provided it is a valid, pre-existing *group_name*.

Multiple privileges may be granted at one time, as long as commas separate them. Do not combine *ALL [PRIVILEGES]* with other privileges, but the remainder may be combined in any order.

PostgreSQL does not support the *WITH* clauses or column-level permissions. Currently, if you want column-level permissions, you must set those up using a view showing only the columns of a table that you want a user to access.

Under PostgreSQL, the creator of an object has all privileges on the object, though they may revoke their own privileges on an object. However, the privilege for the creator of an object to assign privileges on that object cannot be revoked. Similarly, the right to drop an object is permanently assigned to the creator of the object, and cannot be revoked. Users other than the creator of an object do not have permissions on an object until specifically granted.

PostgreSQL's implementation of *GRANT* behaves as if *WITH GRANT OPTION* is always enabled. Any user granted privilege to a table is able to grant their privileges to other users.

PostgreSQL's support for the *GRANT* statement is elementary. In the following examples, *INSERT* privileges are granted on the **publishers** table to the PUBLIC role and *SELECT* and *UPDATE* privileges are granted on the **sales** table to the users **Emily** and **Dylan**:

```
GRANT INSERT ON publishers TO PUBLIC;
```

```
GRANT SELECT, UPDATE ON sales TO emily, dylan;
```

SQL Server

SQL Server's implementation of *GRANT* offers few variations from the ANSI standard. It does not support the ANSI standard's *FROM* clause, nor the *HIERARCHY* or *ADMIN* options. But it does offer the ability to grant specific system privileges and to grant them under another user's security context. The syntax is as follows:

```
GRANT { [object_privilege] [,...] |
```

```
  [system_privilege] }
```

```
[ON { [object] [(column [,...])]
```

```
TO {grantee_name [,...] | role [,...] | PUBLIC | GUEST }
```

```
[WITH GRANT OPTION]
```

```
[AS {group | role}]
```

where:

object_privileges

Grants privileges for a variety of operations, which may all be combined (with the exception of the *ALL [PRIVILEGES]* privilege) in any order, as follows:

ALL [PRIVILEGES]

Grants all privileges currently assigned to the named users and/or for the specified database objects. This is generally not a recommended approach since it can encourage sloppy programming. *ALL* can only be used by members of the *SYSADMIN* and *DB_OWNER* system roles, or by the object owner.

{*SELECT* | *INSERT* | *DELETE* | *UPDATE*}

Grants the specific privilege to a given user on the specified database object, such as a table or view. For column-level privileges, add a parenthetical list of columns of the table. The privileges will then be assigned only for the named columns.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

EXECUTE

Grants the privilege to execute a stored procedure, user-defined function, or extended stored procedure.

system_privileges

Grants one of the following privileges, each giving the ability to execute the corresponding statement:

CREATE DATABASE
CREATE DEFAULT
CREATE FUNCTION
CREATE PROCEDURE
CREATE RULE
CREATE TABLE
CREATE VIEW
BACKUP DATABASE
BACKUP LOG

The privilege to issue a *CREATE* statement also implies the privilege to issue the equivalent *ALTER* or *DROP* statement.

ON { [*object*] [(*column* [,...])] }

Specifies the object for which a privilege is granted. This clause is not required when granting system privileges. If the object is a table or view, you may optionally grant privileges on specific columns. You may grant *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *REFERENCES* privileges on a table or view. You may grant only *SELECT* and *UPDATE* privileges on the columns of a table, view, or a user-defined function. You may grant *EXECUTE* privileges on a stored procedure or extended stored procedure. You may also grant *REFERENCES* privileges on user-defined functions. *REFERENCES* privileges are also required on functions or views created using the *WITH SCHEMABINDING* clause.

TO { *grantee_name* [,...] | *role* [,...] | PUBLIC | GUEST }

Names the user or role who will gain the privilege being granted. Multiple grantees can be listed in a comma-separated list. Use the keyword *PUBLIC* to grant privileges to the *PUBLIC* role, a role that implicitly includes all users. SQL Server also supports the role known as *GUEST*, a role used by all users who do not have any other user account in the database. Because SQL Server allows two different security models (one based on the database and one based on the Windows operating system), you may be granting privileges to a SQL Server user, a Windows user, a Windows group, or a SQL Server database role. The grantee name may be any of these.

WITH GRANT OPTION

Enables the grantee to further grant the given permission to other users or roles. You may use *WITH GRANT OPTION* only when granting object permissions.

AS {*group* | *role*}

Specifies an alternative user or group that has the authority in the current database to execute the *GRANT* command. You can use the *AS* clause to grant privileges as if the granting session is part of a different group or role context from the current one.

Privileges may not be granted in any database other than the current database, nor may privileges be granted in more than one database at a time.



SQL Server has a somewhat different security model than the other platforms described here (and somewhat different from the ANSI standard). As with the ANSI standard, SQL Server uses the *GRANT* command to assign specific privileges to a user or role, and the *REVOKE* command to remove those permissions. SQL Server augments these commands with the *DENY* command, using essentially the same syntax as *REVOKE*.

with the *DENY* command, using essentially the same syntax as *REVOKE*.

DENY, under SQL Server, allows DBAs to declare certain privileges as strictly off-limits to a user or role. Any privileges that have been denied must be revoked before they can be granted. As such, *DENY* takes precedence over *GRANT* and *REVOKE*. *DENY* can be used to effectively take away any permission from an individual that might otherwise inherit permissions from membership in any Windows groups or SQL Server database roles.

SQL Server applies precedence to permissions. Thus, if a user has a privilege granted at the user level, but the permission is revoked at the group level (when the user is a member of a group), the permission is revoked at both levels.

SQL Server also contains a number of fixed system roles with preset privileges on both objects and commands. The SQL Server system roles are:

SYSADMIN

Can perform any activity on the server and has access to all objects.

SERVERADMIN

Can set server configuration options as well as shut down and restart the server.

SETUPADMIN

Can start up procedures and linked servers.

SECURITYADMIN

Can read error logs, change passwords, administrate logins, and grant *CREATE DATABASE* permissions.

PROCESSADMIN

Can administrate processes running on the SQL Server.

DBCREATOR

Can create, alter, and drop databases.

DISKADMIN

Can administrate disks and files.

BULKADMIN

Can perform BCP operations and execute the *BULK INSERT* statement (and regular *INSERT* statement) against tables in the database.

The *SYSADMIN* system role can grant any permission in any database on the server. Members of the *DB_OWNER* and *DB_SECURITYADMIN* system roles can grant any permission on any statement or object in databases they own. Members of the *DB_DDLADMIN* system role, the *SYSADMIN* system role, and the database owner may grant system privileges.

Members of the system roles may add other logins or users to their roles, but not via a *GRANT* statement. Instead, to add a user to a system role, you must use a SQL Server system stored procedure called **sp_addsrvrolemember**.

In addition to the server-wide system roles, there are a number of database-wide system roles. That is, the previous system roles have privileges that span every database on the server, whereas the following system roles exist in every database and have privileges within the scope of a specific database, as follows:

DB_OWNER

Can perform all the functions of the other database system roles, plus perform other maintenance and configuration tasks for the database.

DB_ACCESSADMIN

Can add or remove Windows and SQL Server users and groups to the database.

DB_DATAREADER

Can read data from all user tables in the database.

DB_DATAWRITER

Can read, add, change, or delete data from all user tables in the database.

DB_DDLADMIN

Can add, change, and drop objects in the database, as well as run all DDL statements.

DB_SECURITYADMIN

Can administrate SQL Server roles and users as well as statement and object permissions throughout the database.

DB_BACKUPOPERATOR

Can perform backups of the database.

DB_DENYDATAREADER

Denies permission to read data in the database.

DB_DENYDATAWRITER

Denies permission to change data in the database.

Like the server-wide system roles, database-wide system roles cannot be granted to a user via the *GRANT* statement. Instead, you must use a SQL Server system stored procedure called **sp_addrolemember**.

In the following example, the *CREATE DATABASE* and *CREATE TABLE* system privileges are granted to the users **emily** and **sarah**. Next, numerous permissions are granted on the **titles** table to the editors group. The editors are then able to grant permission to others:

```
GRANT CREATE DATABASE, CREATE TABLE TO emily, sarah
```

```
GO
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON titles
```

```
TO editors
```

```
WITH GRANT OPTION
```

```
GO
```

The following example grants permissions to the database user **sam** and the Windows user **jacob**:

```
GRANT CREATE TABLE TO sam, [corporate\jacob]
```

```
GO
```

The following example shows how to grant permissions using the optional *AS* keyword. In this example, the user **emily**

owns the **sales_detail** table and she grants *SELECT* privileges to the **sales_manager** role. The user **kelly**, who is a member of the **sales_manager** role, wants to grant *SELECT* privileges to **sam**, but cannot because the permissions were granted to the **sales_manager** role and not to her explicitly. **kelly** can use the *AS* clause to get around this hurdle:

```
-- Initial grant
```

```
GRANT SELECT ON sales_detail TO sales_manager
```

```
WITH GRANT OPTION
```

```
GO
```

```
-- Kelly passes the privilege to Sam as a member of the
```

```
    sales_manager role
```

```
GRANT SELECT ON sales_detail TO sam AS sales_manager
```

```
GO
```

See Also

REVOKE

IN Operator

The *IN* operator provides a way to delineate a list of values, either explicitly listed or from a subquery, and compare a value against that list in a *WHERE* or *HAVING* clause. In other words, it gives you a way to say "Is value A *in* this list of values?"

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
{WHERE | HAVING | {AND | OR} }
```

```
    value [NOT] IN ({comp_value1, comp_value2 [...] | subquery} )
```

Keywords

```
{WHERE | HAVING | {AND | OR} } value
```

Permitted under either the *WHERE* or *HAVING* clauses. The *IN* comparison may also be a part of a *AND* or *OR* clause in a multicondition *WHERE* or *HAVING* clause. *value* may be of any datatype, but is usually the name of a column of the table referenced by the transaction, or perhaps a host variable when used programmatically.

NOT

Optionally tells the database to look for a result set that contains values that are not in the list.

IN ({ *comp_value1*, *comp_value2* [,...] | *subquery* })

Defines the list of comparative values (hence, *comp_value*) to compare against. Each *comp_value* must be of the same or compatible datatype as the initial *value*. They are also governed by standard datatype rules. For example, string values must be delimited by quotes, while integer values need none. As an alternative to listing specific values, you may use parentheses to enclose a subquery that returns one or more values of a compatible datatype.

In the following example, generated on SQL Server, we look for all employees in the **employee** table of the **HR** database who have a home state of Georgia, Tennessee, Alabama, or Kentucky:

```
SELECT *
FROM hr..employee
WHERE home_state IN ('AL','GA','TN','KY')
```

Similarly, we can look for all **employees** in the **HR** database who are authors in the **PUBS** database:

```
SELECT *
FROM hr..employee
WHERE emp_id IN (SELECT au_id FROM pubs..authors)
```

You can also use the *NOT* keyword to return a result set based upon the absence of a value. In the following case, the company headquarters is located in New York, and many workers commute in from neighboring states. We want to see all such workers:

```
SELECT *
FROM hr..employee
WHERE home_state
NOT IN ('NY','NJ','MA','CT','RI','DE','NH')
```



Note that Oracle, while fully supporting the ANSI functionality, extends the functionality of the *IN* operator by allowing multiple argument matches. For example, the following *SELECT...WHERE ...IN* statement is acceptable on Oracle:

```
SELECT *
FROM hr..employee e
WHERE (e.emp_id, e.emp_dept) IN
( (242, 'sales'), (442, 'mfg'), (747, 'mkt') )
```

See Also

ALL/ANY/SOME
BETWEEN
EXISTS
LIKE
SELECT
SOME/ANY

INSERT Statement

The *INSERT* statement adds rows of data to a table or view.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported, with variations

The *INSERT* statement allows rows to be written to a table through one of several methods:

- The first method is to insert one or more rows using the *DEFAULT* values specified for a column via the *CREATE TABLE* or *ALTER TABLE* statements.
- The second and most common method is to declare the actual values to be inserted into each column of the record.
- The third method, which populates a table with many records at a time, is to insert the result set of a *SELECT* statement.

SQL2003 Syntax

```
INSERT INTO [ONLY] {table_name | view_name} [(column1 [,...])]
[OVERRIDE {SYSTEM | USER} VALUES]
{DEFAULT VALUES | VALUES (value1 [,...]) | SELECT_statement }
```

Keywords

[ONLY]

Used on typed tables only, this optional keyword ensures that the values inserted into *table_name* do not insert into any subtables.

{*table_name* | *view_name*} [(*column1* [,...])]

Declares the updateable target table or view into which the records will be inserted. You must have *INSERT* privileges on the table or, at a minimum, on the columns that will receive the inserted values. If no schema information is included, as in **scott.employee**, then the current schema and user context are assumed. You may optionally include a list of the columns in the target table or view that will receive data.

OVERRIDE {SYSTEM | USER} VALUES

Requires the *SYSTEM* keyword when inserting a literal *value* into a column that would otherwise be given a system-generated value, such as autogenerated sequence number. The *OVERRIDE USER VALUES* clause does the converse, by inserting system-supplied values even if a user has provided literal values to insert.

DEFAULT VALUES

Inserts all values declared via the *DEFAULT* column characteristic on the target table where they exist, and

NULLs where they do not exist. (Of course, the *DEFAULT* column characteristic can specify *NULL* as a value too.) This operation inserts a single record. It might encounter an error depending on how the *PRIMARY KEY* constraint or *UNIQUE* constraint is constructed on the target table, assuming such constraints exist.

VALUES (*value1* [,...])

Specifies the actual values to be inserted into the target table. The number of values must match the exact number of columns in the column list. Furthermore, the values must be datatype- and size-compatible with the columns of the target table. Each value in the value list corresponds to the column with the same ordinal number in the column list. Thus, the first column will get its data from the first value, the second column from the second value, and so on until all columns are satisfied. You may also optionally use the keywords *DEFAULT* to insert a column's default value and *NULL* to insert a null value.

SELECT_statement

Inserts rows retrieved via the *SELECT* statement into the target table or target view. The values retrieved by the fully formed *SELECT* statement's select item list correspond directly to the columns of the column list. The target table or view may not be referenced in the *SELECT* statement's *FROM* or *JOIN* clauses.

Rules at a Glance

You may insert values into tables and into views built upon a single source table. The *INSERT . . . VALUES* statement adds a single row of data to a table using literal values supplied in the statement. In the following example, a new row in the **authors** table is inserted for the author **Jessica Rabbit**:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone,
    address, city, state, zip, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,
    '1717 Main St', NULL, 'CA', '90675', 1)
```

Every column in the table is assigned a specific, literal value except the **phone** column, which is assigned the default value (as defined during the *CREATE TABLE* or *ALTER TABLE* statement), and the **city** column, which is set to *NULL*.

An important concept to remember is that you may skip columns in the table and set them to *NULL*, assuming they allow *NULL* values. Inserts that leave some columns *NULL* are called partial *INSERTs*. Here is a partial *INSERT* that performs the same work as the preceding example:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT, 1)
```

The *INSERT* statement, combined with a nested *SELECT* statement, allows a table to be quickly populated with one or more rows from the result set of the *SELECT* statement. When using *INSERT . . . SELECT* between a target table and a source table, it is important to ensure that the datatypes returned by the *SELECT* statement are compatible with the datatypes in the target table. For example, to load data from the **sales** table into the **new_sales** table:

```
INSERT INTO sales (stor_id, ord_num, ord_date, qty, payterms,
    title_id)
SELECT
    CAST(store_nbr AS CHAR(4)),
    CAST(order_nbr AS VARCHAR(20)),
    order_date,
    quantity,
    SUBSTRING(payment_terms,1,12),
    CAST(title_nbr AS CHAR(1))
FROM new_sales
```

```
WHERE order_date >= 01-JAN-2005
```

```
-- retrieve only the newer records
```

You must specify the columns in the table that receive data by enclosing their names in parentheses, in a comma-delimited list. This **column_list** can be omitted, but all columns that are defined for the table are then assumed, in their ordinal position. Any column with an omitted value will be assigned its default value (according to any *DEFAULT* column setting on the table) or, if no *DEFAULT* column setting exists, *NULL*. The columns in the column list may be in any order, but you may not repeat any column in the list. Furthermore, the columns and their corresponding *value* entries must agree in terms of datatype and size.

The first of the following two examples leaves off the column list, while the second uses only *DEFAULT* values:

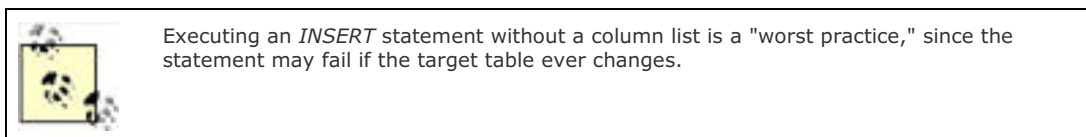
```
INSERT INTO authors
```

```
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,  
       '1717 Main St', NULL, 'CA', '90675', 1)
```

```
INSERT INTO temp_details
```

```
DEFAULT VALUES
```

The first statement would succeed only as long as all the values in the value list correspond correctly with the datatypes and size limitations of the columns of the target table. Any inconsistencies would generate an error. The second statement would succeed only as long as defaults were declared for the columns of the target table, or those columns allowed *NULL* values.



Programming Tips and Gotchas

INSERT statements will *always* fail under the following circumstances:

- When a datatype mismatch occurs between a column and its value
- When a column is defined as not *NULL* and the insertion value is *NULL*
- When a duplicate value is inserted into a *UNIQUE* or *PRIMARY KEY* constraint
- When the inserted values do not meet the requirements of a *CHECK* constraint
- When an inserted value is constrained by a *FOREIGN KEY* constraint because the value is not derived from the declared primary key of another table

The most common error encountered when executing *INSERT* statements is a mismatch between the number of columns and the number of values. Accidentally leave out a value that corresponds to a column, and you are likely to encounter an error that will cause the statement to fail.

INSERT statements also fail when an inserted value is of a datatype that is a mismatch with the column of the target table. For example, an attempt to insert a string like "Hello world" into an integer column would fail. On the other hand, some database platforms automatically and implicitly convert certain datatypes. For example, SQL Server will automatically convert a date value to a character string for insertion into a *VARCHAR* column.

Another common problem encountered with the *INSERT* statement is a size mismatch between a value and its target column. For example, inserting a long string into a *CHAR(5)* target column, or inserting a very large integer into a *TINYINT* column, can cause problems. Depending on the platform you are using, the size mismatch may cause an outright error and rollback of the transaction or the database server may simply trim the extra data. Either result is undesirable. Similarly, a problem can arise when an *INSERT* statement attempts to insert a *NULL* value into a target column that does not accept *NULL*.



Most problems with *INSERT* statements occur because the programmer does not know the target table very well. Make sure you understand the target table or view before writing elaborate *INSERT* statements.

DB2

DB2 supports the ANSI standard form of *INSERT*, with the exception of the *OVERRIDE* clause, and includes some added syntax to control transaction isolation settings:

```
INSERT INTO table_name [(column1 [...])]
{VALUES | VALUES (value1 [...]) | SELECT_statement }
[ WITH { RR | RS | CS | UR }
```

where:

```
INSERT INTO table_name [(column1 [...]) ]
```

Identifies the target table, view, or nickname on the DB2 server. The target table or view may be a standard or typed table or view. The view may be based on a *SELECT* statement containing a *UNION ALL* clause, as long as the inserted row matches the check constraint(s) of exactly one table.

WITH

Controls transaction isolation level for the update statement. *RR* is repeatable read, *RS* is read stability, *CS* is cursor stability, and *UR* is uncommitted read. See *SET TRANSACTION ISOLATION LEVEL* for full details on ANSI transaction isolation levels.

DB2 strictly enforces these rules:

- Values inserted must be compatible with the column datatype
- Integer and string values must not exceed the storage capability of the target column
- Inserted values must pass any check or unique constraints
- Values inserted into a foreign key column must exist in a corresponding primary key elsewhere in the database

DB2 supports identity columns; that is, columns that automatically generate a monotonically increasing value. When inserting into an identity column, DB2 generates a new value for the identity column. Identity columns are defined as either *GENERATED ALWAYS* or *GENERATED BY DEFAULT* upon creation. If defined as *GENERATED ALWAYS*, you should not specify a value for the column — leave it to default. If defined as *GENERATED BY DEFAULT*, you may insert your own value (explicitly or via a subquery), or you can allow DB2 to generate the value for you. In the following examples, assume the **emp_id** column is an identity column with the *GENERATED ALWAYS* property:

```
INSERT INTO employee (emp_id, emp_lname, emp_fname)
```

```
VALUES (DEFAULT, :var_lname, :var_fname)
```

```
INSERT INTO employee (emp_lname, emp_fname)
```

```
VALUES (:var_lname, :var_fname)
```

Both of the example statements shown above will work without error.

MySQL

MySQL supports several *INSERT* syntax options that engender MySQL's reputation for high speed:

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
[INTO] [[database_name.]owner.] table_name [(column1 [,...])]
{VALUES (value1 [,...]) | SELECT_statement |
SET column1=value1, column2=value2 [,...]}
```

where:

LOW_PRIORITY

Defers the execution of *INSERT* until no other clients are reading from the table. This could result in a long wait. *LOW_PRIORITY* should not be used with MyISAM tables because it disables concurrent inserts.

DELAYED

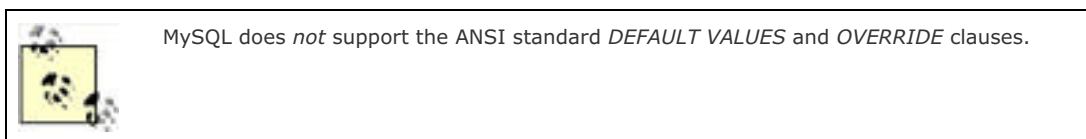
Allows the client to continue at once, even if the *INSERT* has not yet completed.

IGNORE

Tells MySQL not to attempt to insert records that would duplicate a value in a primary key or unique key; otherwise, without this clause, the *INSERT* fails when such duplication occurs. If a duplicate is encountered while the *IGNORE* clause is in use, the duplicate records are ignored while all the other records are inserted.

SET *column=value*

An alternate syntax that allows you to specify values for target columns by name.



MySQL trims any portion of a value that has a size or datatype mismatch. Thus, inserting "10.23 X" into a decimal datatype column will cause "...X" to be trimmed from the inserted value. If you attempt to insert a numeric value into a column that is beyond the range of the column, MySQL will trim the value. Inserting an illegal time or date value in a column will result in a zero value for the target column. Inserting the string "Hello World" into a *CHAR(5)* column will result in the value being trimmed to just the five characters in "Hello". This string trimming feature applies to *CHAR*, *VARCHAR*, *TEXT*, and *BLOB* columns.

Oracle

Oracle's implementation of the *INSERT* statement allows data insertion into a given table, view, partition, subpartition, or object table. It also supports additional extensions such as inserting records into many tables at once and conditional inserts. The syntax is:

-- standard *INSERT* statement

```
INSERT [INTO] {table_name [ [SUB]PARTITION (prt_n_name)] |
(subquery) [WITH {READ ONLY | CHECK OPTION
[CONSTRAINT constr_name ]}] |
TABLE (collection) [ (+) ] } [alias]
[(column1 [,...])]
{VALUES (value1[,...]) [RETURNING expression1 [,...]]
INTO variable1 [,...]} |
```

```
SELECT_statement [WITH {READ ONLY |  
CHECK OPTION [CONSTRAINT constr_name]} }  
-- conditional INSERT statement  
INSERT {[ALL | FIRST]} WHEN condition  
THEN standard_insert_statement  
ELSE standard_insert_statement
```

where:

INSERT [INTO]

Inserts one or more rows into a single table, view, materialized view, or subquery. The *INTO* keyword is optional. You insert a single row using the *VALUES* clause and many rows using a *subquery*.

table_name [[SUB]PARTITION (*prtn_name*)]

Identifies the target into which you will insert data. The target may be a table, view, materialized view, or a subquery. To enable you to fully qualify the target, *table_name* can expand to [*schema*].*table_name*[*@db_link*].

You may optionally identify the schema and remote address (via *@db_link*) of the target, but the current schema and local database are assumed if you do not otherwise specify. You may optionally identify the *PARTITION* or *SUBPARTITION*, through the *prtn_name* parameter, into which the record(s) will be inserted, as long as the target is not an object table or object view.

subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT *constr_name*]}]

Instructs Oracle to insert records into the base table or tables of the *subquery*, where the subquery is a normally formed *SELECT* statement. Essentially, you're using a subquery to construct a view on-the-fly, and the effect is the same as inserting into a view. This is your primary means for inserting values into multiple tables at one time. All of the columns defined by the subquery, across all tables, must have a corresponding value to insert or else a failure will occur. Multitable inserts must use the subquery format. The following options apply to when using subqueries:

WITH READ ONLY

Indicates the subqueried table or view cannot be updated until the statement completes.

*WITH CHECK OPTION [CONSTRAINT *constr_name*]*

Indicates that you cannot insert rows into the table or view that would not pass the *constr_name* check constraint.

TABLE (collection) [(+)] } [alias]

Tells Oracle that the *collection* should be treated like a standard target (i.e., a table or view), whether it be a subquery, a column, a function, or a collection constructor. In any event, the table collection must return a nested table or *VARRAY* set of values. Since constructions can be very long, you can provide an optional *alias*. Aliases are not allowed in multitable insert operations.

[(column1 [,...])]

Specifies the target columns into which data will be inserted. If you leave off the columns, Oracle assumes that the *VALUES* clause or columns of the subquery will perfectly match the columns of the target. Oracle will return an error if you do not insert a value for any columns marked as NOT NULL that do not have a defined, default value.

*VALUES (value1[,...]) [RETURNING *expression1* [,...]] INTO *variable1* [,...]*

Inserts values into the target table or tables. As with the ANSI standard, there must be a matching value for every column, though the value can be *DEFAULT*, or, if the column accepts nulls, the value can be the literal NULL. *DEFAULT* is not allowed when inserting into a view. On multitable insert operations, the *VALUES* clause must return a corresponding value for every item in the select list of the subquery. The syntax is as follows:

RETURNING *expression1*

Retrieves the rows inserted by the operation. The *expression* returned by the statement is often a value being inserted, but could be another value. For example, you might use the returning clause to find the value of an automatically generated primary key. Single row operations store the results into host variables or PL/SQL variables, while multirow operations store them in bind arrays. You can use *RETURNING* against tables, views with a single base table, and materialized views. The *RETURNING* clause is not allowed with multitable insert operations.

INTO *variable1*

Specifies the variables that will hold the values returned as a result of the *RETURNING* clause. You must declare a corresponding host variable or PL/SQL variable for each expression in the *RETURNING* clause. You cannot use the *INTO* clause to hold a *LONG* datatype, with remote objects, on views that have *INSTEAD OF* triggers, or with parallel *INSERT*, *UPDATE*, or *DELETE* statements.

ALL

Performs a multitable *INSERT* statement. *ALL* is used only with the subquery format. Without a *WHEN* clause, *ALL* unconditionally inserts all the data retrieved by the subquery into the tables defined. *ALL*, with a *WHEN* clause, performs conditional insert operations that tell Oracle to evaluate all *WHEN* clauses regardless of the results of any other *WHEN* operation. Each time a *WHEN* clause evaluates as TRUE, Oracle executes the corresponding *INTO* clause. Multitable inserts are not parallelized on index-organized tables or bitmap indexed tables. They are not allowed at all when:

- The target is a view or materialized view
- The target is a remote table
- The *INSERT* command uses a table collection expression
- The table needs more than 999 total target columns
- The subquery uses a sequence

FIRST

Tells Oracle to evaluate the *WHEN* clauses in order and, when it finds the first TRUE expression, execute the corresponding *INTO* clause and skip all other *WHEN* clauses.

WHEN *condition* THEN *standard_insert_statement*

Sets a *condition* and, when the condition is TRUE, executes the *THEN* insert clause. The value of *condition* is evaluated for each column returned in the result set of the subquery. Up to 127 *WHEN* clauses are allowed.

ELSE *standard_insert_statement*

Executes when no *WHEN* clause evaluates as TRUE.

Oracle allows the standard *INSERT* operations as described under the ANSI implementation, such as *INSERT...SELECT* and *INSERT...VALUES*. But it has a great many special variations.

When inserting into tables that have assigned sequences, but sure to use the **sequence_name.nextval** function call to insert the next logical number in the sequence. For example, assume you want to use the **authors_seq** sequence to set the value of **au_id** when inserting a new row into the **authors** table:

```
INSERT authors (au_id, au_lname, au_fname, contract )
```

```
VALUES (authors_seq.nextval, 'Rabbit', 'Jessica', 1)
```

When retrieving values during an *INSERT* operation, check for a one-for-one match between the expressions in the *RETURNING* clause and the variables of the *INTO* clause. The expressions returned by the clause do not necessarily have to be those mentioned in the *VALUES* clause. For example, the following *INSERT* statement places a record into the **sales** table, but places a completely distinct value into a bind variable:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1)
RETURNING hire_date INTO :temp_hr_dt;
```

Notice that the *RETURNING* clause returns the **hire_date** even though **hire_date** is not one of the values listed in the *VALUES* clause. (In this example, it is reasonable to assume a default value was established for the **hire_date** column.)

An unconditional multitable *INSERT* statement into a lookup table that contains a list of all the approved **jobs** in our company looks like this:

```
INSERT ALL
  INTO jobs(job_id, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+1, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+2, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+3, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
SELECT job_identifier, job_title, base_pay, max_pay
FROM   job_descriptions
WHERE  job_status = 'Active';
```

And just to make things more complex, Oracle allows multitable *INSERT* statements that are conditional:

```
INSERT ALL
  WHEN job_status = 'Active'   INTO jobs
  WHEN job_status = 'Inactive' INTO jobs_old
  WHEN job_status = 'Terminated' INTO jobs_cancelled
  ELSE INTO jobs
SELECT job_identifier, job_title, base_pay, max_pay
FROM   job_descriptions;
```

Note that in the above example, you would have to follow each *INTO* clause with a *VALUES* clause if you were skipping *NOT NULL* columns in the target table. The following example shows this syntax:

```
INSERT FIRST
  WHEN job_status = 'Active'
  INTO jobs
  VALUES(job_id, job_desc, min_lvl, max_lvl)
```



```
WHEN job_status = 'Inactive'

    INTO jobs_old

    VALUES(job_id, job_desc, min_lvl, max_lvl)

WHEN job_status = 'Terminated'

    INTO jobs_cancelled

    VALUES(job_id, job_desc, min_lvl, max_lvl)

WHEN job_status = 'Terminated'

    INTO jobs_outsourced

    VALUES(job_id, job_desc, min_lvl, max_lvl)

ELSE INTO jobs

    VALUES(job_id, job_desc, min_lvl, max_lvl)

SELECT job_identifier, job_title, base_pay, max_pay

FROM job_descriptions;
```

Notice that in the above example, the *FIRST* clause also tells Oracle to execute the first occurrence of `job_status = 'Terminated'` by inserting the records into the **jobs_cancelled** table and skipping the **jobs_outsourced** *INSERT* operation.

Oracle allows you to insert data into a table, partition, or view (also known as the target) using a regular or *direct-path INSERT* statement. Under a regular insert, Oracle maintains referential integrity and reuses free space in the target. Under direct-path, Oracle appends data at the end of the target table without filling in any free space gaps elsewhere in the table. This method bypasses the buffer cache and writes directly to the datafiles; hence the term "direct-path."



Oracle allows the use of hints to circumvent default query optimization for *INSERT* statements. For example, you can use the *APPEND* hint to ensure that an *INSERT* uses a direct-path insert. Refer to the platform documentation for more details on hints that are usable with *INSERT*.

Direct-path enhances performance on long, multirecord insert operations. However, the following is a list of restrictions that cause an *INSERT* statement to perform a regular *INSERT*, instead of direct-path:

- Altering the data in the target with an *UPDATE* or *DELETE* statement before the *INSERT* statement in a single transaction. (*UPDATE* and *DELETE* are allowed after the direct-path *INSERT* statement.)
- The *INSERT* statement is or may become distributed.
- The target contains a *LOB* or object datatype column.
- The target has a clustered index or index-organized table.
- The target has triggers or referential integrity constraints.
- The target is replicated.
- The *ROW_LOCKING* initialization parameter is set to *INTENT*.

In addition, Oracle will not allow you to query (e.g., using *SELECT*) a table later in the same transaction after you perform a direct-path *INSERT* into that table until a *COMMIT* has been performed.

Oracle allows you to use parallel direct-path inserts into multiple tables, but you may only use subqueries to insert the data into the tables, not a standard *VALUES* clause.



Inserting *LOBs* and *BFILES* is tricky. You should initialize such values to *NULL* before inserting. *RAW* columns are also tricky. If you insert a regular string into a *RAW* column, all future queries against the column will be forced to use a table scan.

PostgreSQL

PostgreSQL supports the ANSI standard for the *INSERT* statement. It does not support the ANSI SQL clause *OVERRIDE SYSTEM GENERATED VALUES*, but is otherwise identical in syntax and behavior. Refer to [SQL2003 Syntax](#), earlier, for the syntax and usage.



PostgreSQL attempts to perform automatic datatype coercion when the expressions in the *VALUE* clause or the select item list of a subquery do not match the datatypes defined for the target table or view.

SQL Server

SQL Server supports a few extensions to the ANSI standard for *INSERT*. It supports several rowset functions (explained below), as well as the capability to insert the results from stored procedures and extended procedures directly into the target table. SQL Server's syntax is:

```
INSERT [INTO] table_name [(column1 [,...])  
{[DEFAULT] VALUES |  
VALUES (variable1 [,...]) |  
SELECT statement |  
EXEC[UTE] proc_name { [[@param =] value {[OUTPUT]} [,... ] ] }
```

where:

INSERT INTO table_name

Tells SQL Server that the target is either a table, a view, or a rowset function. When inserting into a view, an *INSERT* cannot affect more than one of the base tables in the view, if there are more than one. Rowset functions allow SQL Server to source data from special or external data sources such as XML streams, full-text search file structures (a special structure in SQL Server used to store things like MS Word documents and MS PowerPoint slideshows within the database), or external data sources (like an MS-Excel spreadsheet). Examples are shown later in this section. SQL Server currently supports the following *rowset_functions* for the *INSERT* statement:

OPENQUERY

Executes a pass-through *INSERT* against a linked server. This is an effective means of performing a nested *INSERT* against a data source that is external to SQL Server. The data source must first be declared as a linked server.

OPENROWSET

Executes a pass-through *INSERT* statement against an external data source. This is similar to *OPENDATASOURCE*, except that *OPENDATASOURCE* only opens the data source, it does not actually pass through an *INSERT* statement. *OPENROWSET* is intended for occasional, ad hoc usage only.

column1 [,...]

Identifies one or more columns in the target table. The list must be enclosed in parentheses, and commas must separate each item in the list. SQL Server automatically provides values for *IDENTITY* columns, *TIMESTAMP* columns, and columns with *DEFAULT* constraints.

DEFAULT

Tells the *INSERT* statement simply to create a new record using all of the default values specified for the target table.

```
EXEC[UTE] proc_name { [[@param =] value {[OUTPUT]} [,... ] ] }
```

Tells SQL Server to execute a dynamic Transact-SQL statement, a stored procedure, a Remote Procedure Call (RPC), or an extended stored procedure and store the results in a local table. *Proc_name* is the name of the stored procedure you wish to execute. You may optionally include any of the parameters of the stored procedure, as identified by *@param* (the at sign is required), assign a *value* to the parameter, and optionally designate the parameter as an *OUTPUT* parameter. The columns returned by the result set must match the datatype of the columns in the target table.

Although SQL Server automatically assigns values to *IDENTITY* columns and *TIMESTAMP* columns, it does not do so for *UNIQUEIDENTIFIER* columns. With either of the former datatype columns, you may simply skip them in the *columns* and *values* lists. However, you cannot do that with a *UNIQUEIDENTIFIER* column. You must use the *NEWID*() function to obtain and insert a globally unique ID (GUID):

```
INSERT INTO guid_sample (global_ID, sample_text, sample_int)
```

```
VALUES (NEWID( ), 'insert first record','10000')
```

```
GO
```

When migrating between platforms, remember that inserting an empty string (") into a SQL Server *TEXT* or *VARCHAR* column results in a zero-length string. This is not the same as a null value, as some platforms interpret it.

An example of the *INSERT...EXEC* statements, below, first creates a temporary table called **#ins_exec_container**. Then, the *INSERT* operation retrieves a listing of the *c:\temp* directory and stores it in the temporary table, while the second *INSERT* executes a dynamic *SELECT* statement:

```
CREATE TABLE #ins_exec_container (result_text
```

```
    VARCHAR(300) NULL)
```

```
GO
```

```
INSERT INTO #ins_exec_container
```

```
EXEC master..xp_cmdshell "dir c:\temp"
```

```
GO
```

```
INSERT INTO sales
```

```
EXECUTE ('SELECT * FROM sales_2002_Q4')
```

```
GO
```

This functionality can be very useful in stored procedures where you wish to build business logic using Transact-SQL stored procedures, for example, to determine the state of objects in the database or outside of the database, and then act on those results using Transact-SQL.



SQL Server allows the use of hints to circumvent default query optimization for *INSERT* statements. This type of tuning is only recommended for the most advanced users. Refer to the vendor documentation for more details on hints that are usable with *INSERT*.

INTERSECT Set Operator

The *INTERSECT* set operator retrieves the rows of two or more queries where the rows of the result sets are identical in

both the first and second (and possibly more) queries. In some ways, *INTERSECT* is a lot like an *INNER JOIN* operation.

INTERSECT is in a class of keywords called *set operators*. Other set operators include *EXCEPT* and *UNION*. All set operators are used to simultaneously manipulate the result set of two or more queries; hence the term "set operators."

Platform	Command
DB2	Supported with limitations
MySQL	Not supported
Oracle	Supported with limitations
PostgreSQL	Supported with limitations
SQL Server	Not supported

SQL2003 Syntax

There is technically no limit to the number of queries that you may combine with the *INTERSECT* set operator. The general syntax is:

```
<SELECT statement1>
INTERSECT [ALL | DISTINCT]
[CORRESPONDING [BY (column1, column2, ...)] ]
<SELECT statement2>
INTERSECT [ALL | DISTINCT]
[CORRESPONDING [BY (column1, column2, ...)] ]
...
```

Keywords

ALL

Includes duplicate rows from all result sets.

DISTINCT

Drops duplicate rows from all result sets prior to the *INTERSECT* comparison. Columns containing a NULL value are considered duplicates. (If neither *ALL* nor *DISTINCT* are used, *DISTINCT* behavior is the default.)

CORRESPONDING

Specifies that only columns with the same name in both queries are returned, even though both queries use the asterisk shortcut.

BY

Specifies that only the named columns are returned, even if more columns with corresponding names exist in the queries. Must be used with the *CORRESPONDING* keyword.

Rules at a Glance

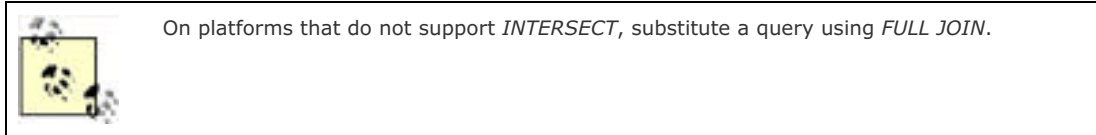
There is only one significant rule to remember when using *INTERSECT*:

The order and number of columns must be the same in all queries. The datatypes of corresponding columns must also be compatible.

The datatypes do not have to be identical, but they should be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest of each column in each ordinal position.

Programming Tips and Gotchas

None of the platforms support the ANSI *CORRESPONDING* [*BY column1, column2, ...*] clause.



The ANSI standard evaluates *INTERSECT* as higher priority than other set operators, while some platforms vary in how they evaluate set operator precedence. You can explicitly control the precedence of set operators using parentheses. Otherwise, the DBMS might evaluate them from leftmost to rightmost expression or from first to last expression.

According to the standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. In order to avoid column and table ambiguity, be sure to alias each column of each table with the same respective alias. For example:

```
SELECT a.au_lname AS 'lastname', a.au_fname AS 'firstname'

FROM authors AS a

INTERSECT

SELECT e.emp_lname AS 'lastname', e.emp_fname AS 'firstname'

FROM employees AS e

ORDER BY lastname, firstname
```

Also, while your column datatypes may be compatible throughout the queries in the *INTERSECT*, there may be some variation in behavior across the DBMS platforms with regard to varying length of the columns. For example, if the **au_lname** column in the first query is markedly longer than the **emp_lname** column in the second query, each platform may apply different rules as to which length is used for the final result. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the columns across the tables have different names. In general, the column names from the first query are used.

DB2

DB2 supports the *INTERSECT* and *INTERSECT ALL* keywords of the ANSI standard, plus the *VALUES* extension:

```
{SELECT statement1 | VALUES (expr1 [,...]) }

INTERSECT [ALL]

[CORRESPONDING [BY (column1, column2, ...)] ]

{SELECT statement2 | VALUES (expr2 [,...]) }

INTERSECT [ALL]

[CORRESPONDING [BY (column1, column2, ...)] ]

...
```

Although ANSI's *INTERSECT DISTINCT* is not supported, *INTERSECT* is the functional equivalent. The *CORRESPONDING* clause is not supported.

When using *INTERSECT* (but not *INTERSECT ALL*), the datatypes *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK* and structured types are not usable.

When the same column name is used across queries, then that name is used in the final result set. If, however, the queries use different names for a column, then DB2 generates a new column name for the final result. That column is then unusable in the *ORDER BY* and *FOR UPDATE* clauses.

If multiple set operations are used in a single query, those enclosed in parentheses are evaluated first. Then, the queries are evaluated in left-to-right order. However, all *INTERSECT* operations are performed before any *UNION* and *EXCEPT* operations. For example:

```
SELECT empno
FROM employee
WHERE workdept LIKE 'E%'
INTERSECT
(SELECT empno
 FROM emp_act
 WHERE projno IN ('IF1000', 'IF2000', 'AD3110'))
UNION
VALUES ('AA0001'), ('AB0002'), ('AC0003')
```

The example above retrieves all employee IDs from the **employee** table for employees who are in any department that starts with "E." However, employee IDs are returned only if they also exist in the employee accounts table, named **emp_act**, or have projects of **IF1000**, **IF2000**, or **AD3110**.

MySQL

Not supported.

Oracle

Oracle supports the *INTERSECT* and *INTERSECT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
INTERSECT
<SELECT statement2>
INTERSECT
...
```

Oracle does not support the *CORRESPONDING* clause. The *INTERSECT DISTINCT* is not supported, but *INTERSECT* is the functional equivalent. Oracle *does not* support *INTERSECT* on queries under the following circumstances:

- Queries with columns of *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatypes
- Queries contain *FOR UPDATE* clause or *TABLE* collection expressions

If the first query in the set operation contains any expressions in the select item list, then include the *AS* keyword to associate an alias with the column resulting from the expression. Also, only the first query in the set operation may contain an *ORDER BY* clause.

For example, you could find all store IDs that also have sales:

```
SELECT stor_id FROM stores
INTERSECT
SELECT stor_id FROM sales
```

PostgreSQL

PostgreSQL supports the *INTERSECT* and *INTERSECT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
```

```
INTERSECT [ALL]
```

```
<SELECT statement2>
```

```
INTERSECT [ALL]
```

```
...
```

PostgreSQL *does not* support *INTERSECT* or *INTERSECT ALL* on queries with a *FOR UPDATE* clause. PostgreSQL does not support the *CORRESPONDING* clause. The *INTERSECT DISTINCT* is not supported, but *INTERSECT* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *INTERSECT* or *INTERSECT ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the rightmost occurrence of *ORDER BY* or *LIMIT* will be assumed to apply to the entire set operation. For example, you could find all authors who are also employees and whose last last name starts with "P":

```
SELECT a.au_lname
```

```
FROM authors AS a
```

```
WHERE a.au_lname LIKE 'P%'
```

```
INTERSECT
```

```
SELECT e.lname
```

```
FROM employee AS e
```

```
WHERE e.lname LIKE 'W%';
```

SQL Server

Not supported. However, you can use *NOT IN* or *NOT EXISTS* operations in conjunction with a correlated subquery as alternatives. Refer to the [EXCEPT Set Operator](#) section for examples.

See also

```
EXCEPT  
SELECT  
UNION
```

IS Operator

The *IS* operator determines whether a value is NULL or not.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

{WHERE | {AND | OR} } *expression* IS [NOT] NULL

Keywords

WHERE | {AND | OR} *expression* IS NULL

Returns a Boolean value of TRUE if the *expression* is NULL, and FALSE if the *expression* is not NULL. The *expression* evaluated for NULL can be preceded by the *WHERE* keyword or *AND* or *OR* keywords.

NOT

Inverses the predicate. The statement will instead return a Boolean TRUE if the value of *expression* is not NULL, and FALSE if the value of *expression* is NULL.

Rules at a Glance

Because the value of NULL is unknown, you cannot use comparison expressions to determine if a value is NULL. For example, the expressions *X = NULL* and *X <> NULL* cannot be resolved because no value can equal, or not equal, an unknown.

Instead, you must use the *IS NULL* operator. Be sure that you do not put the word NULL within quote marks, because if you do that, the DBMS will interpret the value as the word "NULL" and not the special value of NULL.

Programming Tips and Gotchas

Some platforms support the use of a comparison operator to determine if an expression is NULL. However, all platforms covered by this book now support the ANSI *IS [NOT] NULL* syntax.

Sometimes, checking for NULL will make your *WHERE* clause only slightly more complex. For example, rather than a simple predicate to test the value of **stor_id**:

```
SELECT stor_id, ord_date
FROM sales
WHERE stor_id IN (6630, 7708)
```

you need to add a second predicate to accommodate the possibility that **stor_id** might be NULL:

```
SELECT stor_id, ord_date
FROM sales
WHERE stor_id IN (6630, 7708)
OR stor_id IS NULL
```

See Also

```
SELECT
WHERE
```

JOIN Subclause

The *JOIN* subclause enables you to retrieve rows from two or more logically related tables. You are able to define many different join conditions and types of joins, though the types of joins supported by the different platforms vary greatly.

Platform	Command
DB2	Supported, with limitations
MySQL	Supported, with variations
Oracle	Supported
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL2003 Syntax

```
FROM table [AS alias] {CROSS JOIN |  
{ [NATURAL] [join_type] JOIN joined_table [AS alias]  
  { ON join_condition1 [{AND|OR} join_condition2] [...] } |  
  USING (column1 [,...]) } }  
[...]
```

Keywords

FROM *table*

Defines the first table or view in the join.

NATURAL

Specifies that the join (either inner or outer) should be assumed on the tables using all columns of identical name shared between the two tables. Consequently, you should *not* specify join conditions using the *ON* or *USING* clauses. The query will fail if you issue a natural join on two tables that do not contain any columns with the same name(s).

join_type JOIN *joined_table*

Specifies the type of *JOIN* and the second (and any subsequent) table in the join. You may also define an *alias* on any of the *joined_tables*. The join types are:

CROSS JOIN

Specifies the complete cross product of two tables. For each record in the first table, all the records in the second table are joined, creating a *huge* result set. This command has the same effect as leaving off the join condition, and is also known as a "Cartesian Product." *Cross joins are not advisable or recommended.*

[*INNER*] *JOIN*

Specifies that unmatched rows in either table of the join should be discarded. If no join type is explicitly defined in the ANSI style, then this is the default.

LEFT [*OUTER*] *JOIN*

Specifies that all records be returned from the table on the left side of the join statement. If a record returned from the left table has no matching record in the table on the right side of the join, it is still returned. Columns from the right table return NULL values when there is no matching row. It is a good idea to configure all your outer joins as left outer joins, rather than mixing left and right outer joins, wherever possible for consistency.

RIGHT [OUTER] JOIN

Specifies that all records be returned from the table on the right side of the join statement, even if the table on the left has no matching record. Columns from the left table return NULL values when there is no matching row.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

UNION JOIN

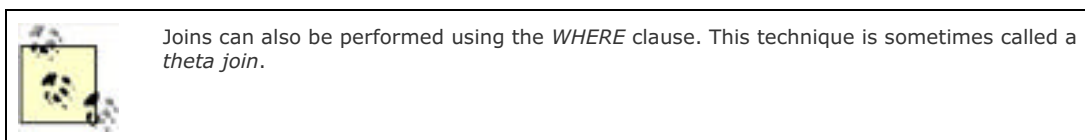
Specifies that all the columns of both tables and every row of both table will be returned. Any columns that have no value in the corresponding joined table are assigned a NULL value.

AS alias

Specifies an alias or shorthand for the joined table. The *AS* keyword is optional when specifying an alias.

ON join_condition

Joins together the rows of the table shown in the *FROM* clause to the rows of the table declared in the *JOIN* clause. You may have multiple *JOIN* statements, all based on a common set of values. These values are usually contained in columns of the same name and datatype appearing in both tables being joined. These columns, or possibly a single column from each table, are called the *join key* or *common key*. Most—but not all—of the time, the join key is the primary key of one table and a foreign key in another table. As long as the data in the columns match, the join can be performed.



Join_conditions are syntactically depicted in the following form (note that join types are intentionally excluded in this example):

FROM table_name1

JOIN table_name2

ON table_name1.column1 = table_name2.column2

[{AND|OR} *table_name1.column3 = table_name2.column4*]

[...]

JOIN table_name3

ON table_name1.columnA = table_name3.columnA

[{AND|OR} *table_name1.column3 = table_name2.column4*]

[...]

[JOIN...]

Use the *AND* operator and the *OR* operator to issue a *JOIN* with multiple conditions. It is also a good idea to use brackets around each pair of joined tables if more than two tables are involved. This makes reading the query much easier.

USING (*column* [,...])

Assumes an equality condition on one or more named *columns* that appear in both tables. The column (or columns) must exist, as named, in both tables. Writing a *USING* clause is a little quicker than writing *...ON table1.columnA = table2.columnA*, but the results are functionally equivalent.

Rules at a Glance

Joins enable you to retrieve records from two (or more) logically related tables in a single result set. You can use an ANSI *JOIN* (detailed here) to perform this operation, or something called a *theta join*. Theta joins, which use the *WHERE* clause to establish the filtering criteria, are the "old" way to do join operations.

For example, you might have a table called **employee** that tracks information about everyone employed in your company. The **employee** table, however, doesn't have extensive information about the job an employee holds. Instead, the **employee** table holds only a **job_id**. All information about the job, such as **description** and **title**, are stored in a table called **job**. Using a *JOIN*, you can easily return columns from both tables in a single set of records. The following sample queries illustrate the difference between a theta and an ANSI *JOIN*:

```
/* Theta join */
```

```
SELECT emp_lname, emp_fname, job_title
```

```
FROM employee, jobs
```

```
WHERE employee.job_id = jobs.job_id;
```

```
/* ANSI join */
```

```
SELECT emp_lname, emp_fname, job_title
```

```
FROM employee
```

```
JOIN jobs ON employee.job_id = jobs.job_id;
```

Whenever you reference multiple columns in a single query, the columns must be unambiguous. In other words, the columns must either be unique to each table, or they must be referenced with a table identifier. In the example above, both tables have a **job_id** column, so references to **job_id** must be qualified with a table identifier. The columns in the query that don't exist in both tables don't need to be qualified by table identifiers. However, queries like this are often very hard to read. The following variation of the previous ANSI join is in better form, because it uses the short, and easy to read, alias **e** to refer to the table:

```
SELECT e.emp_lname, e.emp_fname, j.job_title
```

```
FROM employee AS e
```

```
JOIN jobs AS j ON e.job_id = j.job_id;
```

Previous examples were limited to equi-joins — joins using equality and an equal sign (=). However, most other comparison operators are allowed. You can perform joins on *>*, *<*, *>=*, *<=*, *<>*, and so forth.

You cannot join on large object binary datatypes (i.e., *BLOB*) or other large object datatypes (e.g., *CLOB*, *NLOB*, etc.). Other datatypes are usually allowed in a join comparison.

Following are examples of each type of join:

CROSS JOIN

Following are some cross join examples. The first is a theta join that simply leaves off the join conditions. The second is written using the *CROSS JOIN* clause. The final query is similar in concept to the first, with a *JOIN* clause that omits the join conditions.

```
SELECT *  
  
FROM employee, jobs;
```

```
SELECT *  
  
FROM employee  
CROSS JOIN jobs;
```

```
SELECT *  
  
FROM employee  
JOIN jobs;
```

INNER JOIN

Following is an inner join written using the newer syntax:

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
INNER JOIN publishers AS p ON a.city = p.city  
ORDER BY a.au_lname DESC
```

There are lots of authors in the **authors** table, but very few of them have a city that matches the city of the publishers in the **publishers** table. For example, the above query executed in the **pubs** database on a SQL Server produces results like the following:

first name	last name	publisher
Carson	Cheryl	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems

The join is called an *inner join* because only those records that meet the join condition in both tables are said to be "inside" the join. You could also issue the same query, on platforms that support it, by substituting the *USING* clause for the *ON* clause:

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
INNER JOIN publishers AS p USING (city)  
ORDER BY a.au_lname DESC
```

The results for the above query would be the same.

LEFT [OUTER] JOIN

It is a good idea to configure all your outer joins as left outer joins, for greater consistency, rather than mixing left and right outer joins.

In the following example, we can show a *LEFT OUTER JOIN* by asking for the publisher for each author (we could also substitute the *USING* clause for the *ON* clause, as shown in the example for *INNER JOIN* shown earlier):

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
LEFT OUTER JOIN publishers AS p ON a.city = p.city  
ORDER BY a.au_lname DESC
```

In the above example, every author will be returned along with the publisher's name, where there is a match, or a NULL value, where there is no match. For example, in the SQL Server **pubs** database, the query above returns:

first name	last name	publisher
Yokomoto	Akiko	NULL
White	Johnson	NULL
Stringer	Dirk	NULL
Straight	Dean	NULL
...		

All of the data is returned in the lefthand columns (those from the **authors** table) and any column on the righthand side (those from the **publishers** table) that does not have a match returns a NULL value. The result set shows NULL values where no data matches in the join.

RIGHT [OUTER] JOIN

RIGHT OUTER JOIN is essentially the same as a *LEFT OUTER JOIN*, except everything is weighted toward the table on the right side of the query. For example, the following query executed in the **pubs** database on a SQL Server:

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
RIGHT OUTER JOIN publishers AS p ON a.city = p.city  
ORDER BY a.au_lname DESC
```

returns the following result set:

first name	last name	publisher
Carson	Cheryl	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems
NULL	NULL	New Moon Books
NULL	NULL	Binnet & Hardley

All of the data is returned on the righthand side of the query (hence the term "right outer join") and any columns from the left-hand side that do not have a match return a NULL value. The result set shows NULL values where no data matches in the join.

NATURAL [INNER | {LEFT | RIGHT}[OUTER]] JOIN

Natural joins are a substitute for the *ON* or *USING* clause, so do not use *NATURAL* with the *ON* or *USING* clauses. For example:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors AS a
NATURAL RIGHT OUTER JOIN publishers AS p
ORDER BY a.au_lname DESC
```

The preceding query will work as the earlier examples, but only if both tables possess a column called **city** and that is the only column that they hold in common. You could similarly perform any of the other types of joins (*INNER*, *FULL*, *OUTER*) using the *NATURAL* prefix.

FULL [OUTER] JOIN

Specifies that all rows from either table be returned, regardless of matching records in the other table. The result set shows NULL values, where no data matches in the join. Note that the *OUTER* keyword is optional. If we take our earlier example query and render it as a *FULL JOIN*, it looks like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors AS a
FULL JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

The result set returned by the query is actually the accumulation of the result sets of issuing separate *INNER*, *LEFT*, and *RIGHT* join queries (some records have been excluded for brevity):

first name	last name	publisher
Yokomoto	Akiko	NULL
White	Johnson	NULL
Stringer	Dirk	NULL

...

Dull	Ann	NULL
del Castillo	Innes	NULL
DeFrance	Michel	NULL
Carson	Cheryl	Algodata Infosystems
Blotchet-Halls	Reginald	NULL
Bennet	Abraham	Algodata Infosystems
NULL	NULL	Binnet & Hardley
NULL	NULL	Five Lakes Publishin
NULL	NULL	New Moon Books
...		
NULL	NULL	Scootney Books
NULL	NULL	Ramona Publishers
NULL	NULL	GGG&G

As you can see above, with the *FULL JOIN* you get some records with the NULLs on the right and data on the left (*LEFT JOIN*), some with all data (*INNER JOIN*), and some with NULLs on the left and data on the right (*RIGHT JOIN*).

Programming Tips and Gotchas

If an explicit *join_type* is omitted, then an *INNER JOIN* is assumed. Note that there are many types of joins, each with their own rules and behaviors, described in the "Rules at a Glance."

In general, you should favor the *JOIN* clause over the *WHERE* clause for describing join expressions. This not only keeps your code cleaner, making it easy to differentiate join conditions from search conditions, but you also avoid the possibility of buggy behavior resulting from some platform-specific implementations of outer joins specified using the *WHERE* clause.

In general, we do not recommend the use of labor-saving keywords like *NATURAL*, since the subclause will not automatically update itself when the structure of the underlying tables change. Consequently, statements using these constructs may fail when a table change is introduced without also changing the query.

Not all join types are supported by all platforms, so refer to the sections below for full details on platform-specific join support.



Joins involving more than two tables can be difficult. When joins involve three or more tables, it is a good idea to think of the query as a series of two table joins.

DB2

DB2 supports *INNER* and *OUTER* joins using the *ON* condition clause. It does not support *CROSS* or *NATURAL* join syntax, nor the *USING* clause. DB2's JOIN clause syntax is:

```
FROM table [AS alias]
```

```
{ {[INNER] | [ {LEFT | RIGHT | FULL} [OUTER]] }
```

```
JOIN joined_table [AS alias]
```

```
{ ON join_condition1 [{AND|OR} join_condition2 [ ... ] }
```

```
[...]
```

Refer to the "Rules at a Glance" section for examples.

MySQL

MySQL supports most ANSI syntax, except that natural joins are only supported on outer joins, not on inner joins. MySQL's *JOIN* syntax is:

```
FROM table [AS alias]
{[STRAIGHT_JOIN joined_table] |
{ {[INNER] | | [CROSS] |
  [NATURAL] [ {LEFT | RIGHT | FULL} [OUTER]]}
  JOIN joined_table [AS alias]
  { ON join_condition1 [{AND|OR} join_condition2] [...] ] |
  USING (column1 [,...]) } } }
[...]
```

where:

STRAIGHT_JOIN

Forces the optimizer to join tables in the exact order they appear in the *FROM* clause.

The *STRAIGHT_JOIN* keyword is functionally equivalent to *JOIN*, except that it forces the join order from left to right. This option was supplied because MySQL might, rarely, join the tables in the wrong order.

Refer to the [Rules at a Glance](#) section for examples.

Oracle

Oracle fully supports the entire ANSI standard *JOIN* syntax. However, Oracle only began to support ANSI standard *JOIN* syntax in Version 9. Consequently, any older Oracle SQL code exclusively uses *WHERE* clause joins. Oracle's old syntax for outer theta joins included adding (+) to column names on the side *opposite* of the outer table. Oracle's old syntax for outer joins comes from the fact that the table supplying the null value rows in effect has null value rows added to it. Oracle represents this by appending the expression "(+)" to the column name on the opposite side of the direction of the join.

For example, the following query does a *RIGHT OUTER JOIN* on the **authors** and **publishers** tables. The old Oracle syntax looks like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors a, publishers p
WHERE a.city(+) = p.city
ORDER BY a.au_lname DESC
```

while the same query in the ANSI syntax would look like:


```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors AS a
RIGHT OUTER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

Refer to the [Rules at a Glance](#) section for more *JOIN* examples.

Oracle is unique in offering partitioned outer joins, which are useful for filling gaps in result sets due to sparse data storage. For example, assume we store production records in a manufacturing table keyed on day and **product_id**. The table holds a row showing quantity produced for each product for any day on which it is made, but there are no rows for the days it is not produced. This is considered sparse data, since a list of all rows will not show every day for every product. For calculation and reporting purposes, it's very useful to create result sets where each product has a row for every day, whether or not it was manufactured. Partition outer join makes it simple to do that, since it lets you define a logical partition and apply an outer join to each partition value. Here is the manufacturing example, which does a partitioned outer join with a **times** table to make sure each **product_id** has the full set of dates in a specified time range:

```
SELECT times.time_id AS time, product_id AS id, quantity AS qty
FROM manufacturing
PARTITION BY (product_id)
RIGHT OUTER JOIN times
ON (manufacturing.time_id = times.time_id)
WHERE manufacturing.time_id
BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
      AND TO_DATE('06/10/05', 'DD/MM/YY')
ORDER BY 2, 1;
```

Following is the output from this query:

time	id	qty
01-OCT-05	101	10
02-OCT-05	101	
03-OCT-05	101	
04-OCT-05	101	17
05-OCT-05	101	23
06-OCT-05	101	
01-OCT-05	102	
02-OCT-05	102	
03-OCT-05	102	43
04-OCT-05	102	99
05-OCT-05	102	
06-OCT-05	102	87

To get these results without using a partitioned outer join would require more complex and less efficient SQL.

PostgreSQL

PostgreSQL fully supports the ANSI standard. Refer to the [Rules at a Glance](#) section for examples.

SQL Server

SQL Server supports *INNER*, *OUTER*, and *CROSS* joins using the *ON* clause. SQL Server does not support *NATURAL* join syntax, nor the *USING* clause. SQL Server's *JOIN* syntax is:

```
FROM table [AS alias]  
  
{ {[INNER] | [CROSS] | [ {LEFT | RIGHT | FULL} [OUTER]] }  
  
  JOIN joined_table [AS alias]  
  
    { ON join_condition1 [{AND|OR}  
  
      join_condition2] [...] }  
  
[...]
```

Refer to the [Rules at a Glance](#) section for examples.

See Also

```
SELECT  
WHERE
```

LIKE Operator

The *LIKE* operator enables specified string patterns in *SELECT*, *INSERT*, *UPDATE*, and *DELETE* statements to be matched, specifically in the *WHERE* clause. A specified pattern may include special wildcard characters. The specific wildcards supported vary from platform to platform.

Platform	Command
DB2	Supported
MySQL	Supported, with variations
Oracle	Supported
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
WHERE expression [NOT] LIKE string_pattern  
  
[ESCAPE escape_sequence]
```

Keywords

WHERE *expression* LIKE

Returns a Boolean TRUE when the value of *expression* matches the *string_pattern*. The expression may be a column, a constant, a host variable, a scalar function, or a concatenation of any of these. It should not be a user-defined type, nor should it be certain types of *LOBs*.

NOT

Inverses the predicate. The statement returns a Boolean TRUE if the value of *expression* does not contain the *string_pattern*, and FALSE if the value of *expression* contains the *string_pattern*.

ESCAPE *escape_sequence*

Allows you to search for the presence of characters that would normally be interpreted as wildcards.

Rules at a Glance

Matching string patterns is easy with *LIKE*, but there are a couple of simple rules to remember:

- All characters, including trailing and leading spaces, are important.
- Differing datatypes may be compared using *LIKE*, but they store string patterns differently. In particular, be aware of the differences between *CHAR*, *VARCHAR*, and *DATE* datatypes.
- Using *LIKE* may negate indexes or force the DBMS to use alternative, less optimal indexes than a straight comparison operation.

The ANSI standard currently supports two wildcard operators that are supported by all of the platforms covered in this book:

%

Matches any string.

_ (*underscore*)

Matches any single character.

The first query in the following example retrieves any city record with "ville" in its name. The second query returns authors with a first name *not* like Sheryl or Cheryl (or Aheryl, Bheryl, Dheryl, Zheryl, and so forth).

```
SELECT * FROM authors
```

```
WHERE city LIKE '%ville%';
```

```
SELECT * FROM authors
```

```
WHERE au_fname NOT LIKE '_heryl';
```

Some of the platforms support additional wildcard symbols. These are described in the platform-specific sections that follow.

Use of the *ESCAPE* clause allows you to look for wildcard characters in the strings stored in your database. Using this mechanism, you designate a character as your escape character, typically a character that does not otherwise appear in the pattern string. For example, you might designate the tilde (~) because you know it never appears in the pattern string. Any wildcard character preceded by the escape sequence is then not treated as a wildcard, but rather as the character itself. For example, to look through the **comments** column of the **sales_detail** table (on SQL Server) to see whether any customers have mentioned a newly introduced discount:

```
SELECT ord_id, comment
```

```
FROM sales_detail
```

```
WHERE comment LIKE '%~%' ESCAPE '~'
```

In this case, the first and last % are interpreted as wildcards. But the second % character is interpreted as just that, a % character, because it is preceded by the designated escape sequence.

Programming Tips and Gotchas

The usefulness of *LIKE* is based on the wildcard operators that it supports. *LIKE* returns a Boolean TRUE value when the comparison finds one or more matching values.

The default case sensitivity of the DBMS is very important to the behavior of *LIKE*. For example, Microsoft SQL Server is not case-sensitive by default (though it can be configured that way). Thus, Microsoft SQL Server will evaluate the strings *DAD* and *dad* to be equal. Oracle, on the other hand, is case-sensitive. Thus, on Oracle, a comparison of *DAD* and *dad* would show them to be unequal. Here's an example query to better illustrate this point:

```
SELECT *  
  
FROM authors  
  
WHERE Iname LIKE 'LARS%'
```

This query on Microsoft SQL Server would find **authors** whose last names are stored as 'larsen' or 'lars,' even though the search was for uppercase 'LARS%'. Oracle would not find 'Larsen' or 'Lars', because Oracle performs case-sensitive comparisons.

DB2

DB2 supports the ANSI SQL2003 syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It supports *ESCAPE* sequences.

DB2 is case sensitive, so its implementation of *LIKE* is completely case sensitive. You should use a function like *UPPER* or *TRANSLATE* to ensure that differently cased values are always evaluated uniformly. In addition, DB2 will implicitly convert the code page of either the *string_pattern* or the *escape_sequence* to that of the *expression*, as long as none of them are defined as *FOR BIT DATA*.

MySQL

MySQL supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It also supports the *ESCAPE* clause.

MySQL also supports the special functions of *REGEXP* and *RLIKE*, plus *NOT REGEXP* and *NOT RLIKE* for the evaluation of regular expressions. MySQL, after Version 3.23.4, is not case sensitive by default.

Oracle

Oracle supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It supports *ESCAPE* sequences. Oracle's *LIKE* syntax is as follows:

```
WHERE expression [NOT] {LIKE | LIKEC | LIKE2 |  
  
    LIKE4} string_pattern  
  
[ESCAPE escape_sequence]
```

The Oracle-specific syntax elements have the following meanings:

LIKEC

Uses UNICODE complete characters.

LIKE2

Uses UNICODE USC2 codepoints.

LIKE4

Uses UNICODE UCS4 codepoints.

Since Oracle is case sensitive, you should enclose the *expression*, the *string_pattern*, or both, with the *UPPER* function. That way, you are always comparing apples to apples.

PostgreSQL

PostgreSQL supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It supports *ESCAPE* sequences.

PostgreSQL is case sensitive by default. PostgreSQL provides the keyword *ILIKE* for case-insensitive pattern matching. You can also use the operator *~~* as an equivalent to *LIKE*, and *~~** for *ILIKE*, as well as *!~~* and *!~~** for *NOT LIKE* and *NOT ILIKE*, respectively. These are all extensions to the ANSI SQL2003 standard.

For example, the following queries are functionally the same:

```
SELECT * FROM authors
WHERE city LIKE '%ville';

SELECT * FROM authors
WHERE city ~~ '%ville';
```

Since these queries are in lowercase, you might run into a case-sensitivity problem. That is, the queries are looking for a lowercase '%ville', but the table might contain uppercase (and unequal) values of 'BROWNSVILLE', 'NASHVILLE', and 'HUNTSVILLE'. You can get around this as shown below:

```
-- convert the values to upper case

SELECT * FROM authors
WHERE city LIKE UPPER('%ville');

-- perform the pattern match using case insensitivity

SELECT * FROM authors
WHERE city ~~* '%ville';

SELECT * FROM authors
WHERE city ILIKE '%ville';
```

Although beyond the scope of this text, you should be aware that PostgreSQL also supports POSIX regular expressions. See the platform documentation for details.

SQL Server

SQL Server supports the ANSI syntax for *LIKE*. It supports *ESCAPE* sequences. It supports the following additional wildcard operators:

[]

Matches any value in the specified set, as in [abc], or any range, as in [k-n].

[^]

Matches any characters not in the specified set or range.

Using SQL Server's additional wildcard operators, you have some added capabilities. For example, you can retrieve any author with a last name like Carson, Carsen, Karson, or Karsen:

```
SELECT * FROM authors
WHERE au_lname LIKE '[CK]ars[eo]n'
```

Or you can retrieve any author with a last name that ends in "arson" or "arsen," *but not* Larsen or Larson:

SELECT * FROM authors

WHERE au_lname LIKE '[A-Z^L]ars[eo]n'



Remember that when performing string comparisons with *LIKE*, all characters in the pattern string are significant, including all leading and trailing blank spaces.

See Also

SELECT, UPDATE, DELETE
WHERE

MERGE Statement

The *MERGE* statement is sort of like a *CASE* statement for DML operations. It combines *UPDATE* and *INSERT* statements into a single atomic statement with either/or functionality.

Basically, *MERGE* examines the records of a source table and a target table. If the records exist in both the source and target tables, based upon predefined conditions, then records in the target table are updated with the values of the records in the source table. If records do not exist in the target table that do exist in the source table, then they are inserted into the target table. The *MERGE* statement was added in the SQL2003 release of the ANSI standard.

Platform	Command
DB2	Supported
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
MERGE INTO {object_name | subquery} [ [AS] alias ]  
USING table_reference [ [AS] alias ]  
ON search_condition  
WHEN MATCHED  
    THEN UPDATE SET column = { expression | DEFAULT } [, ...]  
WHEN NOT MATCHED  
    THEN INSERT [( column [, ...] )] VALUES ( expression [, ...] )
```

Keywords

MERGE INTO {object_name | subquery}

Declares the target object of the merge operation. The target object may be a table or updateable view of *object_name*, or the target object may be a nested table subquery rather than an explicitly declared table or updateable view.

[AS] *alias*

Provides an optional alias for the target table.

USING *table_reference*

Declares the source table, view, or subquery of the merge operation.

ON *search_condition*

Specifies the condition or conditions on which a match between the source and target table are evaluated. The syntax is essentially the same as the *ON* subclause of the *JOIN* clause. For example, when merging records from the **new_hire_emp** table into the **emp** table the clause might look like *ON emp.emp_id = new_hire_emp.emp_id*.

WHEN MATCHED THEN UPDATE SET *column* = { *expression* | DEFAULT } [, ...]

Declares that if a record from the source table has a matching record in the target table (based on the *search_condition*), then one or more specified *columns* of the target table are updated with the indicated value of *expression*.

WHEN NOT MATCHED THEN INSERT [(*column* [, ...])] VALUES (*expression* [, ...]

Declares that if a record from the source table does not have a matching record in the target table (based on the *search_condition*), then a new record is inserted into the target table using one or more specified *columns* using the value of *expression*.

Rules at a Glance

The rules for using *MERGE* are straightforward:

- The *WHEN MATCHED* and *WHEN NOT MATCHED* clauses are required, but may not be specified more than once.
- The target table can be a standard updateable table, an updateable view, or an updateable subquery.
- If the *table_reference* is a subquery, enclose it in parentheses.
- The *search_condition* clause should not contain any references to stored procedures or user-defined functions.
- The *search_condition* clause may contain multiple elements using the *AND* or *OR* operators.
- If the column list is omitted from the *WHEN NOT MATCHED* clause, then a column list of all the columns in the target table, in ordinal position, is assumed.
- Other important rules used by the *MERGE* statement are self-evident. For example, the columns referenced in the *WHEN MATCHED* clause must be updateable.

Programming Tips and Gotchas

The *MERGE* statement has sometimes been nicknamed the "upsert" statement. That is because it allows, in a single operation, a set of records to be either inserted into a table or, if they already exist, updated with new values.

The only tricky aspect of the *MERGE* statement is simply getting used to the idea of either/or processing of the *INSERT* and *UPDATE* statements.

Assume that we have two tables. The **EMP** table contains all employees of the company who have successfully completed the mandatory 90-day probationary period at the start of their employment. Employees in the **EMP** table can also have several statuses, such as active, inactive, and terminated. Any new hire to the company is recorded in the **NEW_HIRE** table. After 90 days, they are moved into the **EMP** table like all other regular employees. However, since

our company hires college interns every summer, it's very likely that some of our new hires will actually have a record in the **EMP** table from last year with a status of inactive. Using pseudocode, the business problem is summarized as:

```
For each record in the NEW_HIRE table
    Find the corresponding record in the EMP table
    If the record exists in the EMP table
        Update existing data in the EMP table
    Else
        Insert this record into the EMP table
    End If
End For
```

We could write a rather lengthy stored procedure that will examine all the records of the **NEW_HIRE** table and then conditionally perform *INSERT* statements for the entirely new employees or *UPDATE* statements for the returning college interns. But the following ANSI standard *MERGE* statement makes this process much easier:

```
MERGE INTO emp AS e
    USING (SELECT * FROM new_hire) AS n
    ON e.empno = n.empno
WHEN MATCHED THEN
    UPDATE SET
        e.ename = n.ename,
        e.sal   = n.sal,
        e.mgr   = n.mgr,
        e.deptno = n.deptno
WHEN NOT MATCHED THEN
    INSERT ( e.empno, e.ename, e.sal, e.mgr, e.deptno )
    VALUES ( n.empno, n.ename, n.sal, n.mgr, n.deptno );
```

As you can see, the *MERGE* statement is very useful for data loading operations.

DB2

DB2 supports the ANSI standard for the *MERGE* statement.

MySQL

Not supported.

Oracle

Oracle supports the *MERGE* statement with only the tiniest variations, which are almost entirely evident by comparing the Oracle syntax diagram against the ANSI standard syntax diagram:

```
MERGE INTO [schema.] {object_name | subquery} [alias ]
USING [schema.]table_reference [alias]
ON ( search_condition )
```


WHEN MATCHED THEN

```
UPDATE SET column = { expression | DEFAULT } [, ...]
```

WHEN NOT MATCHED THEN

```
INSERT ( column [, ...] ) VALUES ( expression [, ...] )
```

The differences between the ANSI standard and Oracle's implementation include:

- The Oracle implementation does not allow the *AS* keyword with assigning an alias to the target or source table.
- Oracle requires parentheses around the *search_condition* clause.
- The *WHEN NOT MATCHED* clause requires an insert column list, while the ANSI standard makes it optional.

Refer to the examples in the earlier "Rules at a Glance" and "Programming Tips and Gotchas" for more information.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

INSERT
JOIN
SELECT
SUBQUERY
UPDATE

OPEN Statement

The *OPEN* statement is one of four commands used in cursor processing, along with *DECLARE*, *FETCH*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than in a complete set. The *OPEN* statement opens a pre-existing server cursor created with the *DECLARE CURSOR* statement.

Cursors are especially important on relational databases because databases are set-based, while most client-centric programming languages are row-based. So cursors allow programmers and databases to perform operations a single row at a time, while the default behavior of a relational database is to operate on a whole set of records.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported

SQL2003 Syntax

OPEN cursor_name

Keywords

OPEN cursor_name

Identifies and opens the previously defined cursor created with the *DECLARE CURSOR* command.

Rules at a Glance

At the highest level, a cursor must be:

1. Created using *DECLARE*
2. Opened using *OPEN*
3. Operated against using *FETCH*
4. Dismissed using *CLOSE*

By following these steps, you create a result set similar to that of a *SELECT* statement, except that you can operate against each individual row within the result set. For example, assume, on a DB2 database, that you've created a cursor called **employee_cursor**, containing three columns:

```
DECLARE CURSOR employee_cursor FOR
```

```
SELECT lname, fname, emp_id
```

```
FROM employee
```

```
WHERE hire_date >= 'FEB-14-2004';
```

Having created the cursor, you can now open the cursor and fetch data from it into some host variables:

```
OPEN employee_cursor;
```

```
FETCH NEXT FROM employee_cursor
```

```
INTO :emp_last_name, :emp_first_name, :emp_id
```

The *FETCH* statement allows you to position the current row within the result set retrieved by the the cursor. If you haven't accessed the cursor yet, you will be given the first record in the result set. You can then use any procedural code you want to operate against the values in that row.

The following generic SQL example opens a cursor and fetches all the author firstname and author lastname values from the **authors** table:

```
DECLARE employee_cursor CURSOR FOR
```

```
SELECT au_lname, au_fname
```

```
FROM pubs.dbo.authors
```

```
WHERE lname LIKE 'K%'
```

```
OPEN employee_cursor
```

```
FETCH NEXT FROM employee_cursor
```

```
BEGIN
```

```
    FETCH NEXT FROM employee_Cursor
```

```
END
```

```
CLOSE employee_cursor
```

Programming Tips and Gotchas

The most common error encountered with the *OPEN* statement is failing to close the cursor properly. Although the *OPEN* statement is detailed in isolation here, it should always be managed as a group with the *DECLARE*, *FETCH*, and *CLOSE* statements. Although you will not get an error message for failing to close a cursor, the cursor may continue to consume memory, locks, and other resources on the server as long as it is open. For example, if you forget to close cursors, you could be creating a headache somewhat similar to a memory leak — each cursor consumes memory until it is closed. Even if you no longer use a cursor, it is still taking up memory that the database server might otherwise be using elsewhere. So take a little extra time to make sure that every declared and opened cursor is eventually closed.

Cursors are often used in stored procedures and in batches of procedural code. They are useful when you need to perform actions on individual rows rather than on entire sets of data at a time. But because cursors operate on individual rows and not on sets of data, they are often much slower than other means of accessing data. Just make sure that you perform strong analysis on your approach. Many challenges, such as a convoluted *DELETE* operation or a very complex *UPDATE*, can be solved by using clever *WHERE* and *JOIN* clauses instead of cursors.

DB2

DB2 supports the ANSI standard for *OPEN* with a couple of additions:

```
OPEN cursor_name [USING {variable_name1 [,...]} | DESCRIPTOR descriptor_name]
```

The DB2 specific syntax elements are:

```
USING {variable_name1 [,...]} | DESCRIPTOR descriptor_name
```

Supplies a list of variables for any parameters in the prepared *SELECT* statement. If the *DECLARE CURSOR* statement used a prepared *SELECT* statement with variables, you cannot use the *USING* clause. (If you use *USING* and there are no parameters, then *USING* is ignored.) The *USING* clause elements are as follows:

```
variable_name1 [,...]
```

Specifies a host variable or variables. As you would expect, each variable corresponds to a parameter of the same ordinal position. DB2 locator variables and file reference variables are allowable, in addition to host variables. Host variables are used for static SQL.

```
DESCRIPTOR descriptor_name
```

Identifies a descriptor area (SQLDA) containing a valid description of the dynamic SQL variable(s). Descriptor variables are used for dynamic SQL.

For example, the following DB2 code declares several host (called **hostvarxxx**) and dynamic variables (defined with the *PREPARE* statement), and then uses them in a cursor called **dynamic_cursor**:

```
EXEC SQL BEGIN DECLARE SECTION;

    static short hostvar_int;

    char hostvar_vchar64[65];

    char statement_str[200];

EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE statement_name

    FROM :statement_str;

EXEC SQL DECLARE dynamic_cursor CURSOR

    FOR statement_name;

EXEC SQL OPEN dynamic_cursor

    USING :hostvar_int, :hostvar_vchar64;
```

MySQL

Not supported.

Oracle

Oracle fully supports the ANSI standard. Plus, Oracle allows parameters to be passed directly into the cursor when it is opened. Do this using the following format:

```
OPEN cursor_name parameter1 [...]
```

PostgreSQL

PostgreSQL does not have an *OPEN CURSOR* statement. Instead, PostgreSQL implicitly opens a cursor when it is created by the *DECLARE* statement.

SQL Server

In addition to the standard *OPEN* statement, SQL Server allows global cursors using the following syntax:

```
OPEN [GLOBAL] cursor_name
```

where:

cursor_name

Specifies the name of the cursor, or is a string variable containing the name of the cursor, created earlier with the *DECLARE CURSOR* statement.

GLOBAL

Enables the cursor to be referenced by multiple users, even if they have not explicitly been assigned permissions to the cursor. Otherwise, a local cursor is assumed.

SQL Server allows you to declare several different kinds of cursors. If a cursor is *INSENSITIVE* or *STATIC*, then the *OPEN* statement creates a temporary table to hold the result set of the cursor. Similarly, if the declared cursor has the *KEYSET* option, then a temporary table is automatically created to hold the keyset.

See Also

CLOSE CURSOR
DECLARE CURSOR
FETCH
SELECT

ORDER BY Clause

The *ORDER BY* clause specifies the sort order of the result set retrieved by a *SELECT* statement.

Platform	Command
DB2	Supported, with limitations
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL2003 Syntax

ORDER BY {*sort_expression* [*COLLATE collation_name*]
[*ASC* | *DESC*]} [,...]

Keywords

ORDER BY

Specifies the order in which rows should be returned by a query. You should not anticipate a specific ordering if you exclude the *ORDER BY* clause, even if you specify a *GROUP BY* clause and it appears that a sort has been done.

sort_expression

Specifies an item in the query that will help determine the order of the result set. You can have multiple *sort_expressions*, and they are usually column names or column aliases from the query. However, they may also be expressions like (*salary * 1.02*). SQL92 allowed the use of ordinal positions for *sort_expressions*. This functionality has been deprecated, and should not be used in SQL2003 queries.

COLLATE collation_name

Overrides the default collation of the *sort_expression*, and applies the *collation_name* to the sort expression for the purposes of the *ORDER BY* clause.

[*ASC* | *DESC*]

Specifies that the *sort_expression* should be returned in either ascending order (*ASC*) or descending order (*DESC*).

Rules at a Glance

The *ORDER BY* clause should reference columns as they appear in the select item list of the *SELECT* statement, preferably using their alias, if one exists. For example:

```
SELECT au_fname AS first_name, au_lname AS last_name
FROM authors
ORDER BY first_name, last_name
```

The *ORDER BY* clause uses a major-to-minor sort ordering. This means that the result set is ordered by the first column referenced, equal values in the first column are then ordered by the second column, equal values in the second column are then ordered by the third column, and so forth.

The individual aspects of a column's ordering — *COLLATE* and *ASC/DESC* — are independent of the other columns in the *ORDER BY* clause. Thus, you could order a result set in ascending order by one column, and then flip the next column and order it in descending order:

```
SELECT au_fname AS first_name, au_lname AS last_name
FROM authors
ORDER BY au_lname ASC, au_fname DESC
```

NULLs are always grouped together (i.e., considered equal) for the purposes of sorting. Depending on your platform, NULLs will be clumped together at the top or at the bottom of the result set. The following query on SQL Server:

```
SELECT title, price
FROM titles
ORDER BY price, title
```

provides this result set (edited for brevity):

title	price
Net Etiquette	NULL
The Psychology of Computer Cooking	NULL
The Gourmet Microwave	2.9900
You Can Combat Computer Stress!	2.9900
Life Without Fear	7.0000
Onions, Leeks, and Garlic: Cooking Secrets of the Me	20.9500
Computer Phobic AND Non-Phobic Individuals: Behavior	21.5900
But Is It User Friendly?	22.9500

You can force NULLs to appear at the top or bottom of the result set using *ASC* or *DESC*. Of course, all the non-NULL rows of the result set are also ordered in ascending or descending order too.

The ANSI standard for *ORDER BY* also supports the use of columns for the *sort_expression* that are not referenced in the select item list. For example, the following query is valid under SQL2003:

```
SELECT title, price
FROM titles
ORDER BY title_id
```

Looking at the example above, you can see that although the query does not select **title_id**, column is the primary *sort_expression*. The result set is returned in **title_id** order even though that column is not selected.

Programming Tips and Gotchas

When using set operators (*UNION*, *EXCEPT*, *INTERSECT*), only the last query may have an *ORDER BY* clause. You should not use *ORDER BY* in subqueries of any type.

A number of behaviors that were supported in SQL92 are deprecated in SQL2003. You should avoid these usages:

References to table aliases

For example, *ORDER BY e.emp_id* should be changed to *ORDER BY emp_id*. If there is an ambiguous column name, then use an alias to compensate.

References to ordinal position

Use explicitly defined column aliases to compensate.

You may sort not only on columns, but also on expressions involving columns or even literals:

```
SELECT SUBSTRING(title,1,55) AS title, (price * 1.15) as price
FROM titles
WHERE price BETWEEN 2 and 19
ORDER BY price, title
```

When sorting on expressions from the select item list, you should use aliases to make the *ORDER BY sort_expression* column references easier!

DB2

DB2 supports the ANSI syntax, except for the *COLLATE* option. It also has the *ORDER OF* extension:

```
ORDER BY {ORDER OF table_name | sort_expression
        [{ASC | DESC}] }
```

where:

```
ORDER OF table_name
```

Tells DB2 to order the rows of a parent table as they are ordered in the subquery. To understand this concept more fully, remember that DB2 supports ordering at the subquery level. However, the ordered subquery does not return ordered result set unless the *ORDER BY ORDER OF* clause is used. The *table_name* must be the name of a table referenced in the parent query. For example:

```
SELECT qty, price, order_nbr
FROM
  (SELECT order_nbr, qty, price FROM sales_Q1_2005
  UNION
  SELECT order_nbr, qty, price FROM sales_Q2_2005
  ORDER BY price, qty, order_nbr) AS ytd_sales
ORDER BY ORDER OF ytd_sales
```

DB2 still supports SQL92 functions, such as ordering by ordinal position. By default, DB2 sorts NULL values as higher than all other values.

Another example of order subqueries follows. First, we will create a view called **best** that returns the top five best selling books from the **sales** table, and then we will find the top five authors in order:

```
CREATE VIEW best(title) AS
  SELECT title FROM sales
  ORDER BY sold
  FETCH FIRST 5 ROWS ONLY
SELECT author
FROM best JOIN authors ON best.title = author.title
ORDER BY ORDER OF best
```

DB2 *does not* support *ORDER BY* on columns of the following datatypes: *LONG VARCHAR*, *CLOB*, *LONG VARGRAPHIC*, *DBCLOB*, *BLOB*, and *DATALINK*. DB2 also does not support ordering by structure types.

MySQL

MySQL supports the ANSI standard, except for the *COLLATE* option.

You should not attempt to *ORDER BY* columns of the *BLOB* datatype because only the first bytes, defined by the *max_sort_length* setting, are used in the sort. By default, MySQL sorts NULL values as lowest (first) for *ASC* order and highest (last) for *DESC* order.

Oracle

Oracle supports the ANSI syntax, except for the *COLLATE* option. It also supports the *SIBLINGS* and *NULLS {FIRST | LAST}* options. Oracle's *ORDER BY* syntax is:

ORDER [SIBLINGS] BY

{sort_expression [ASC | DESC] [NULLS {FIRST | LAST}]} [...]

where the Oracle-specific keywords are:

ORDER [SIBLINGS] BY *sort_expression*

Sorts the result set of the query in order of the *sort_expressions*. A *sort_expression* may be a column name, an alias, an integer indicating a column's ordinal position, or another expression (e.g., *salary * 1.02*). The *ORDER SIBLINGS BY* clause tells Oracle to preserve any ordering specified by a hierarchical query clause (*CONNECT BY*), and to use the sort expression order for ordering of siblings in the hierarchy.

NULLS {FIRST | LAST}

NULLS FIRST and *NULLS LAST* specify that the records containing NULLs should appear either first or last, respectively. By default, Oracle places NULLs last for ascending order sorts, and first for descending order sorts.

You can emulate the behavior of the *COLLATE* option for a single session by using the *NLSSORT* function with the *NLS_SORT* parameter. You can explicitly emulate the behavior of the *COLLATE* option for all sessions on the server by using the *NLS_SORT* initialization parameter or, implicitly, with the *NLS_LANGUAGE* initialization parameter.

Oracle continues to support deprecated SQL92 features such as sorting by ordinal position. You should *not* perform an *ORDER BY* on any *LOB* column, nested table, or *VARRAY*.

PostgreSQL

PostgreSQL supports the ANSI SQL2003 standard, with the exception of the *COLLATE* option. It also supports the *USING* extension:

ORDER BY *{sort_expression1 [ASC | DESC | USING operator]}* [...]

where:

USING *operator*

Specifies a specific comparison operator. Thus, you may sort by *>*, *<*, *=*, *>=*, *<=*, and so forth. Ascending order is the same as specifying *USING <*, while descending order is the same as *USING >*.

PostgreSQL sorts NULLs as the highest values. Thus, NULLs will appear at the end when sorting in ascending order, and at the beginning when sorting in descending order.

SQL Server

SQL Server supports the ANSI standard including the *COLLATE* option. For example, the following query retrieves author firstnames from the **authors** table in the *SQL_Latin1* collation:


```
SELECT au_fname  
  
FROM authors  
  
ORDER BY au_fname  
  
COLLATE SQL_Latin1_general_cp1_ci_as
```

SQL Server continues to support the deprecated SQL92 features of the *ORDER BY* clause, including the ability to specify *ORDER BY* columns by their ordinal positions.

By default, SQL Server sorts NULL values higher than all other values.

You should not use *TEXT*, *IMAGE*, or *NTEXT* columns as *sort_expressions* on SQL Server.

See Also

SELECT

RELEASE SAVEPOINT Statement

The *RELEASE SAVEPOINT* statement eliminates one or more previously created savepoints in the current transaction.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
RELEASE SAVEPOINT savepoint_name
```

Keywords

savepoint_name

Represents a named savepoint (or target specification) created earlier in the transaction with the *SAVEPOINT* statement. The *savepoint_name* must be unique within the transaction.

Rules at a Glance

Use the *RELEASE SAVEPOINT* statement within a transaction to destroy a named savepoint. Any savepoints that were created after the named savepoint will also be destroyed.

To illustrate the behavior of savepoints, the following example code inserts a few records, creates a savepoint named **first_savepoint**, and then releases it:

```
INSERT authors (au_id, au_lname, au_fname, contract )  
  
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);  
  
SAVEPOINT first_savepoint;
```

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);

INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);

RELEASE SAVEPOINT first_savepoint;

COMMIT;
```

In this example, the **first_savepoint** savepoint is destroyed and then all three records are inserted into the **authors** table.

In the next example, we perform the same action but with more savepoints:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);

SAVEPOINT first_savepoint;

INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);

SAVEPOINT second_savepoint;

INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);

SAVEPOINT third_savepoint;

RELEASE SAVEPOINT second_savepoint;

COMMIT;
```

In the earlier example, when we release the savepoint called **second_savepoint**, the database actually releases **second_savepoint** and **third_savepoint**, since **third_savepoint** was issued after **second_savepoint**.

Once released, a savepoint name can be reused.

Programming Tips and Gotchas

Issuing either a *COMMIT* or full *ROLLBACK* statement will destroy all open savepoints in a transaction. Issuing a *ROLLBACK TO SAVEPOINT* statement returns the transaction to its state at the specified savepoint. Thus, any savepoints declared afterwards are nullified.

DB2

DB2 supports the *RELEASE SAVEPOINT* statement fully with one optional keyword:

```
RELEASE [TO] SAVEPOINT savepoint_name
```

where:

TO

An optional noise word.

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

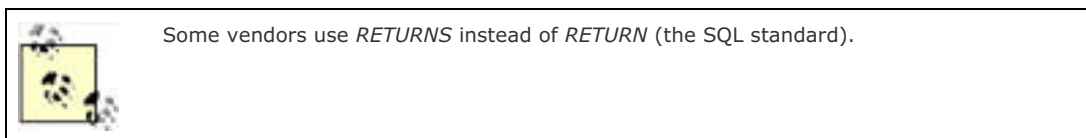
Not supported.

See Also

ROLLBACK
SAVEPOINT

RETURN Statement

The *RETURN* statement terminates processing within a SQL-invoked function (as opposed to a host-invoked function) or stored procedure and returns the routine's result value.



Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Supported, with limitations
SQL Server	Supported

SQL2003 Syntax

RETURN return_parameter_value | NULL

Keywords

return_parameter_value

Represents a value returned by the routine code, and can be any sort of a wide variety of values.

NULL

Terminates the function without returning an actual value

Rules at a Glance

Use the *RETURN* statement within procedural code to terminate processing. For example, you might create a user-defined function that takes a complex and often-used *CASE* expression and, when passed a parameter, returns a single, easy-to-understand expression value.

Programming Tips and Gotchas

Although the *RETURN* statement is categorized as a separate command within SQL, it is deeply intertwined with the *CREATE FUNCTION* and *CREATE PROCEDURE* statements. Consequently, the *RETURN* statement is almost always found embedded in one of these other commands. Check each statement's section in this book, or your vendor documentation, to get a more complete understanding of each platform's implementation of *RETURN* within the context of each statement.

DB2

DB2 uses the *RETURN* statement to exit a routine and, in a procedure, return any integer value. In a user-defined function, the *RETURN* statement returns the actual value that is calculated by the function. DB2 also allows query-based results to be returned by the statement, as in the following:

```
RETURN [ {expression | NULL | WITH table_name  
        ( column [...]) [AS (select_statement) ] } ] select_statement
```

where:

expression

Returns a literal or variable to the client or calling program.

NULL

Returns a null value of the datatype defined in the return clause for the calling user-defined function or method.

WITH *table_name* (*column* [...])

Acts as a *common table expression* (in other words, an ad hoc view) with each *column* keyword defining a column of the subsequent *select_statement*.

AS (*select_statement*) *select_statement*

Specifies the record or records to be returned from the function executing the *RETURN* statement. The number of columns in the record(s) retrieved by the second *select_statement* must match the number of columns in the *WITH* clause. This clause is not allowed in stored procedures.

If *RETURN* is used in a procedure, the value of *expression* must be an integer, but not in a user-defined function. The *NULL* and *WITH columns AS select_statement* clauses can be used with user-defined functions and methods, but not stored procedures. Here's a simple example of *RETURN*:

BEGIN

...<some SQL code>...

GOTO ERROR_HANDLER

... <some SQL code>...

SUCCESS: RETURN 0

ERROR_HANDLER: RETURN -200

END

If you need to check the *RETURN* value of a stored procedure that is called by another stored procedure, use the *GET DIAGNOSTICS* statement to find it. Refer to the DB2 documentation for more information on *GET DIAGNOSTICS*.

MySQL

Not supported. All procedural code in MySQL is written in C or C++. You can use C or C++ programming techniques to return value(s) to the client.

Oracle

Oracle supports the ANSI standard syntax for *RETURN*, excluding the *NULL* keyword. (Oracle does support the return of a *NULL* value, just not using the ANSI syntax.) Oracle allows the *RETURN* clause only in user-defined functions and user-defined operators. The returned value cannot be a *LONG*, *LONG RAW*, or *REF* datatype in the *CREATE OPERATOR* statement. PL/SQL user-defined functions fully support Boolean datatypes internally, but you cannot invoke a Boolean UDF from a SQL statement. Therefore, just use *INT* (0 or 1) or *VARCHAR2* ('TRUE' or 'FALSE') to hold Boolean values.

This example creates a function. The function returns the value that is stored in the **proj_rev** variable to the calling session:

```
CREATE FUNCTION project_revenue (project IN varchar2)
RETURN NUMBER
AS
    proj_rev NUMBER(10,2);
BEGIN
    SELECT SUM(DECODE(action,'COMPLETED',amount,0) -
        SUM(DECODE(action,'STARTED',amount,0) +
        SUM(DECODE(action,'PAYMENT',amount,0)
    INTO proj_rev
    FROM construction_actions
    WHERE project_name = project;
    RETURN (proj_rev);
END;
```

PostgreSQL

PostgreSQL supports the ANSI standard syntax for *RETURN*, excluding the *NULL* keyword:

```
RETURNS return_parameter_value | NULL
```

PostgreSQL allows you to define user-defined functions using either SQL code or C/C++. (PostgreSQL does not currently allow stored procedures, though you can duplicate their functionality with user-defined functions). For the purposes of this discussion, we are only interested in SQL functions.

The output allowed by the *return_parameter_value* may be a base type, complex type, *SETOF* type, *OPAQUE* modifier, or the same as the type of an existing column. The *SETOF type* modifier is used to return a set of items via the *RETURNS* statement, rather than a single data value. The *OPAQUE* modifier indicates the *RETURNS* statement does not return a value. *OPAQUE* can only be used within triggers.

SQL Server

SQL Server supports the *RETURN* statement with this syntax:

```
RETURN [return_integer_value]
```

The *RETURN* command is typically used in stored procedures or user-defined functions. The command causes an

immediate and complete exit from the program and, optionally, returns an integer value upon exit. SQL Server procedures implicitly return a zero value if no *RETURN* clause is supplied in the procedure definition. Any commands that follow the *RETURN* statement are ignored.

The *RETURN* statement in the following function returns an integer representing the calculated value:

```
CREATE FUNCTION metric_volume -  
    Input dimensions in centimeters.  
    (@length decimal(4,1),  
    @width decimal(4,1),  
    @height decimal(4,1) )  
RETURNS decimal(12,3) -- Cubic Centimeters.  
AS  
BEGIN  
    RETURN ( @length * @width * @height )  
END  
GO  
REVOKE
```

This example creates a function that returns a calculated value, the metric volume of a space, to the calling session.

See Also

CREATE/ALTER FUNCTION PROCEDURE
CREATE/ALTER TRIGGER

REVOKE Statement

The *REVOKE* statement takes two main forms. The first form of the statement removes specific statement permissions from a user, group, or role. The second form of the statement removes access permissions to specific database objects or resources.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 syntax for *REVOKE* takes this general form:

```
REVOKE { [special_options] | {privilege [...] | role [...]} }  
ON database_object_name  
FROM grantee_name [...]
```

[GRANTED BY {CURRENT_USER | CURRENT_ROLE}]

{CASCADE | RESTRICT}

Keywords

special_options

Allows the use of one of three optional *special_options*:

GRANT OPTION FOR

Undoes the *WITH GRANT OPTION* privilege assigned to a user, meaning that the user can no longer grant privileges to other users on the object but their own privileges on the object are still intact. (This clause is valid to revoke a privilege, but not to revoke a role.) Refer to the [GRANT Statement](#) section for more details.

HIERARCHY OPTION FOR

Undoes the *WITH HIERARCHY OPTION* privilege that allows a user to *SELECT* not only from the named table, but also all of its subtables. (This clause is valid to revoke a privilege, but not to revoke a role.)

ADMIN OPTION FOR

Undoes the ability to grant a role to other users. (This clause is valid to revoke a privilege, but not to revoke a role.)

privilege

Revokes privileges for a variety of statements, which may all be combined in any order:

ALL PRIVILEGES

Revokes all privileges currently assigned to the named users and/or for the specified database objects. Generally not a recommended approach, since it can encourage sloppy coding.

EXECUTE

Revokes the privileges to execute a routine (i.e., a stored procedure, user-defined function, or method).

{*SELECT* | *INSERT* | *UPDATE* | *DELETE*}

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked for *SELECT*, *INSERT*, and *UPDATE* privileges.

REFERENCES

Revokes the user's privilege to create any constraints or assertions referencing the database object as a parent object. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

TRIGGER

Revokes the user's ability to create triggers on the specified tables. As a side effect of the revoke, any triggers that depend on the privilege are also dropped.

UNDER

Revokes the privilege to create subtypes or typed tables.

USAGE

Revokes privileges to use a domain, user-defined type, character set, collation, or translation.

role

Revokes a specific, pre-defined role from the named grantee identified *FROM* clause. For example, the database administrator might want to create a role called **Reporter** that has read-only access to several tables. When the role is granted to the user, the user is able to do all of the actions the role may do. See the [GRANT Statement](#) section for more details.

ON *database_object_name*

Revokes privileges on a specific, predefined database object identified by the *database_object_name* provided. SQL2003 does not include support for system privileges but many implementations provide a variation of this clause to revoke such privileges. (The *ON* subclause is not used at all when revoking system privileges or roles.) The *database_object_name* may be one of the following:

```
{ [TABLE] object_name | DOMAIN object_name |  
  COLLATION object_name |  
  CHARACTER SET object_name | TRANSLATION object_name |  
  TYPE object_name / [SPECIFIC] {ROUTINE | FUNCTION |  
  PROCEDURE | METHOD}  
  object_name }
```

FROM *grantee_name*

Names the user(s) or role(s) who will lose the specified privilege. The *PUBLIC* keyword may also be used to revoke privileges granted to the *PUBLIC* global user list. Multiple grantees can be listed with a comma between each.

FROM { *CURRENT_USER* | *CURRENT_ROLE* }

Optionally used to specify who granted the privilege to the user in the first place. For example, when using *...FROM CURRENT_USER*, the privilege are revoked only if the current user granted the privilege. Otherwise, the statement would fail. When omitted, *CURRENT_USER* is the default.

RESTRICT | *CASCADE*

Limits a revoke operation to only the specified privilege (*RESTRICT*) or causes the specified privilege and all dependent privileges to be revoked (*CASCADE*), and possibly cause objects that depend on the privilege to be dropped. Note that *REVOKE...RESTRICT* will fail if dependent privileges exist. The dependent privileges must be revoked first.

Rules at a Glance

A specific privilege on a specific database object can be revoked for a single user using *REVOKE privilege_name ON object_name FROM grantee_name RESTRICT*. A specific privilege on a specific object may be revoked from all users via the *PUBLIC* global user list.

When revoking privileges from multiple grantees, simply place a comma between each. You may also revoke privileges from one or more grantees and the *PUBLIC* role in a single *REVOKE* statement. (*PUBLIC* is described in detail under the [GRANT Statement](#) section. When a grant is issued to *PUBLIC*, that means everyone has permissions to the object without the need for a grant specifically to the object.)

When revoking privileges on a table, the privileges may extend to the column level by showing a list of columns enclosed in parentheses after the table name.

Programming Tips and Gotchas

Most of the platforms isolate privileges at the user and role level. (Remember, a *role* is a group of privileges.) Thus, an individual user who is assigned two roles may have an individual set of permissions three times. In this situation, you would have to revoke the user's privileges directly and remove the user from both roles to completely negate their permissions.

An important aspect about *REVOKE* (and its sister command *GRANT*) is that certain elements of the command are geared toward object-level permissions, while other options of the command are more oriented toward roles or administrative privileges. Generally, these operations are never combined. The differences in object-level permissions and administrative privileges are fully explained under each database platform section below. For example, you might want to revoke certain object-level privileges for a given role (**salespeople**) or individual user (**e_fudd** and **prince_edward**):

```
REVOKE SELECT ON TABLE authors FROM salespeople RESTRICT;  
  
REVOKE ALL PRIVILEGES ON TABLE sales FROM e_fudd,  
  
prince_edward CASCADE;
```

A lot of table-level privileges can even be assigned down to the column level. Those privileges can be revoked:

```
REVOKE INSERT(au_id, au_fname, au_lname)  
  
ON authors FROM e_fudd;
```

The special option clauses (*GRANT OPTION*, *HIERARCHY OPTION*, and *ADMIN OPTION*) all exist to allow you to revoke the ability of users to pass privileges or roles on to other users. However, these clauses do not prevent the first group of users themselves from exercising those privileges. For example, we no longer want anyone in the role **manager** to grant their *UPDATE* privileges to other users:

```
REVOKE GRANT OPTION FOR UPDATE ON sales FROM manager CASCADE;
```

You can also remove role privileges from a user with the *REVOKE* command:

```
REVOKE manager FROM e_fudd CASCADE;
```

A common best practice is to keep *REVOKE* and *GRANT* statements as self-contained and logical as possible. Under this practice, you should avoid using the *CASCADE* and *ALL PRIVILEGES* clauses because they perform work for you that may not be immediately evident within the scope of the command.

DB2

DB2 offers a variety of additional options in its implementation of the *REVOKE* command:

```
REVOKE { [database_privileges [,...]] ON DATABASE } |  
  
[CONTROL ON INDEX index_name] |  
  
[ {BIND | CONTROL | EXECUTE} ON PACKAGE package_name] |  
  
[ {ALTERIN | CREATEIN | DROPIN} ON SCHEMA schema_name] |  
  
[PASSTHRU ON SERVER server_name] |  
  
[EXECUTE ON [SPECIFIC] {FUNCTION | PROCEDURE |  
  
METHOD} routine_name] |  
  
[object_privilege [,...]] ON [TABLE] object_name] |  
  
[{ALTER | USAGE} ON SEQUENCE sequence_name] |  
  
[USE OF TABLESPACE tablespace_name] } }  
  
[FROM [ {USER | GROUP} ] { grantee_name [,...]} |
```

group_name [,...] | PUBLIC }]

[RESTRICT]

where:

database_privileges

Revokes one or more privileges that apply to the entire database. The database privileges are:

BINDADD

Grants the authority to create packages.

CONNECT

Grants the authority to access the database. When *CONNECT* is revoked, all other explicitly granted privilege still exist, but are disabled until *CONNECT* privileges are regranted.

CREATETAB

Grants the authority to create tables.

CREATE_NOT_FENCED

Grants the authority to create functions that are not fenced into a specific user process; that is, they may execute within the database manager's process.

DBADM

Grants the superuser authority, meaning the ability to create any object, fully access any object created by other users, and grant those privileges to other users.

IMPLICIT_SCHEMA

Grants the authority to implicitly create a schema, though this privilege doesn't impact the ability to explicitly create a schema or any objects within a schema.

LOAD

Grants the authority to explicitly *LOAD* a database.

CONTROL ON INDEX *index_name*

Revokes the privilege to drop an index, once created, of the specified *index_name*. If you leave off the *FROM* clause, the current user context is assumed.

{ BIND | CONTROL | EXECUTE } ON PACKAGE *package_name*

Revokes the privileges associated with packages. Note that *RUN* and *PROGRAM* are acceptable synonyms for *EXECUTE* and *PACKAGE*, respectively. If you leave off the *FROM* clause, the current user is assumed if all privileges are user-level (GRANTEETYPE of U) or the current group is assumed if all the privileges are group-level (GRANTEETYPE of G). If the privileges are a mix of user- and group-level, an error will result. The package privileges are:

BIND

Grants the authority to bind or rebind the specified package. *CONTROL* privileges must also be revoked,

if the grantee has them, to properly remove *BIND* privileges.

CONTROL

Grants the authority to drop the specified package, as well as the authority to extend privileges to other grantees.

EXECUTE

Grants the authority to execute a package. Like *BIND*, *CONTROL* privileges must also be revoked, if the grantee has them, to properly remove *EXECUTE* privileges.

{ ALTERIN | CREATEIN | DROPIN } ON SCHEMA *schema_name*

Revokes the privileges associated with manipulating a schema (that is, a user-owned set of objects). Unlike some of the other optional clauses of this DB2 command, the *ON SCHEMA* syntax requires the *FROM* clause. Using this command does not supercede any specific privileges granted within a schema. Thus, you might revoke privileges on a user or group using the *ON SCHEMA* clause but the user or group might still be able to do any number of other activities within that schema because of other grant statements.

ALTERIN

Grants the authority to alter or comment on objects in a schema defined by *ON SCHEMA schema_name*.

CREATEIN

Grants the authority to create objects in a schema defined by *ON SCHEMA schema_name*.

DROPIN

Grants the authority to drop objects in the schema defined by *ON SCHEMA schema_name*.

PASSTHRU ON SERVER *server_name*

Revokes the privilege to access an external data source of *server_name* using pass-through mode.

EXECUTE ON

Revokes privileges on an ANSI *routine* (i.e., a user-defined function, user-defined method, or stored procedure). The elements of this clause are as follows:

[SPECIFIC] FUNCTION *function_name*

Specifies that the user loses execution privileges on a specific function. A *SPECIFIC* function name may be specified. DB2 also supports functions without names. In this case, *FUNCTION [(param [,...])]* should be used, including any and all parameters of the function. If *routine_name* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific function name to revoke from a user the execution privilege on all functions in the current or specified schema.

PROCEDURE [*schema.*]*procedure_name*

Specifies that the user loses execute privileges on all stored procedures in the specified schema, even those created in the future. If *schema* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific stored procedure name to revoke the execution privilege from a user on all stored procedures in the current or specified schema.

[SPECIFIC] METHOD *method_name*

Specifies that the user loses rights to execute all methods of *routine_name*, even those that have not yet been created. A *SPECIFIC* method name may be specified. DB2 also supports methods without names. In this case, *METHOD [(param [,...])]* can be used, including any and all parameters of the function. Alternately, the asterisk

(*) can be used to specify that all types in the schema are revoked from the user.

object_privilege

Revokes privileges for a variety of statements on tables, views, and nicknamed tables or views. These may be combined in any order:

ALL [PRIVILEGES]

Revokes all privileges currently assigned to the named users and/or for the specified database objects as a shorthand for every privilege the grantor has available to grant. The *PRIVILEGES* keyword is optional. We do not recommend this approach since it can encourage sloppy programming.

ALTER

Revokes the privilege to add or alter columns, constraints, comments, or triggers on a specified table or view and restricts the ability to change nicknamed column names or datatypes or to change a column option for a nickname.

CONTROL

Revokes the ability to drop the table or view or execute a RUNSTATS utility against it.

INDEX

Revokes the ability to create an index against a table or an index specification on a nickname.

{SELECT | INSERT | UPDATE | DELETE}

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked for the *SELECT*, *INSERT*, and *UPDATE* statements.

REFERENCES

Revokes the user's privilege to create or drop referencing constraints (such as *CHECK* or *FOREIGN KEY*) referencing the database object as a parent object.

*{ALTER | USAGE} ON SEQUENCE *sequence_name**

Revokes privileges on a sequence in an explicit schema or the current schema (*USAGE*), or the privilege to execute the *ALTER SEQUENCE* statement (*ALTER*).

*USE OF TABLESPACE *tablespace_name**

Revokes the *USE* privilege on a tablespace. Note that the user might be able to continue to make tables in the specified tablespace if rights on the tablespace are granted to *PUBLIC*, or if the user is in a group (such as *DBADM*) that also has privileges in the specified tablespace.

*FROM { [USER | GROUP]} *grantee_name**

Names the user or users who will lose the specified privilege. When both a user and group exist with the same name, use the optional *GROUP* or *USER* keywords to control where the privileges will be dropped. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role. Multiple grantees can be listed with a comma between each.

RESTRICT

Identical to the ANSI usage, except that *RESTRICT* is required on *SCHEMA* and *FUNCTION* privileges.

DB2 supports a number of privileges, as described above, through a specific user. In the following example, the user **emily** loses the create table (*CREATETAB*) privilege:

```
REVOKE CREATETAB ON DATABASE FROM emily
```

This revokes the privilege of creating tables from user **emily**.

The following statement revokes the privilege of creating packages from the group **student**. The *FROM GROUP* clause should always be used when a user and group exist with the same name.

```
REVOKE BINDADD ON DATABASE FROM GROUP student
```

The first command in the next example revokes the privilege to drop the index **upkcl_aidind** from the user **dylan** and the group of users called **editors**, while the second command revokes the tablespace privileges of the user **sam**.

```
REVOKE CONTROL ON INDEX upkcl_aidind
```

```
FROM USER dylan, GROUP editors
```

```
REVOKE USE OF TABLESPACE futures FROM USER sam
```

The next example revokes the ability to execute the package **publications** by both the group **editors** and the user **jake**.

```
REVOKE EXECUTE
```

```
ON PACKAGE publications
```

```
FROM GROUP editors, USER jake
```

```
REVOKE CREATEIN ON SCHEMA pubs FROM anna
```

The next example revokes the ability of user **anna** to create new objects in the schema **pubs**, while the second command revokes the privilege to pass through to data source **east**.

```
REVOKE PASSTHRU ON SERVER east FROM USER anna
```

Specific object-level privileges are also easy to grant and revoke. In some cases, you can issue the same command in several ways, depending upon whether you include a lot of the optional keywords. For example, the following two commands do the same thing:

```
REVOKE UPDATE ON employee FROM sam
```

```
REVOKE UPDATE ON TABLE employee FROM USER sam
```

In both of the examples above, the user **sam** loses the privilege to update the **employee** table. You can also combine object-level privileges and users and/or groups in a single command. For example, the following command revokes all the write privileges (*INSERT*, *UPDATE*, *DELETE*) from the users **katie** and **jake**, as well as from the group **editors**:

```
REVOKE INSERT, UPDATE, DELETE ON employee FROM USER katie,
```

```
USER jake, GROUP editors
```

DB2 ships with the *PUBLIC* group, which includes all users. In addition, there are several levels of system privileges on a DB2 installation whose privileges cannot be modified:

SYSADM

Issues system administration privileges (the highest level of authority). Users with this level have all privileges in all databases.

DBADM

Issues database-wide privileges. Users with this level have all privileges in a specific database, though they cannot execute operations that require administrative access to system resources (such as the DB2 statement *CREATE TABLESPACE*).

SYSCtrl

Issues system control privileges without data access privileges. Users with this level have the ability to create, modify, and drop databases and tablespaces, and are able to halt instances or databases.

SYSMaint

Issues system control privileges without data access privileges (at a lower level than *SYSCtrl*). Users with this level have the ability to update database configuration files, issue backups and restores, and monitor a database.

Users can be granted and revoked these system-level privileges, but the privileges of the specific scope of each system-level privilege (system-wide, database-wide, etc.) cannot be altered.

MySQL

The *REVOKE* statement rolls back permissions previously granted to one or more users. MySQL supports many of the standard keywords of the ANSI standard. Notable exceptions for support include *TRIGGER*, *EXECUTE*, and *UNDER*. MySQL also has a nice shortcut to allow global assignment or revocation of privileges. Please refer back to the ANSI standard description for those elements of the command not described below:

```
REVOKE [ { ALL [PRIVILEGES] |
        {SELECT | INSERT | UPDATE} [ (column_name [,...]) ] | DELETE |
        REFERENCES [ (column_name [,...]) ] } |
        [ USAGE ] | [{ALTER | CREATE | DROP}] | [FILE] | [INDEX] | [PROCESS] |
        [RELOAD] | [SHUTDOWN] | [CREATE TEMPORARY TABLES] | [LOCK TABLES] |
        [REPLICATION CLIENT] | [REPLICATION SLAVE] | [SHOW DATABASES] |
        [SUPER] },...]
ON {table_name | * | *.* | database_name.*}
FROM user_name [,...]
```

where:

ALL [PRIVILEGES]

Synonymous with *ALL PRIVILEGES*. On MySQL, this includes all privileges that apply at the level indicated by the *ON* clause, except the *GRANT OPTION*. (This includes privileges like *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and others.)

SELECT | INSERT | UPDATE | DELETE

Revokes the ability to read, write, modify, and erase data from tables (and possibly from specific columns of a table), respectively.

REFERENCES

Unimplemented.

USAGE

Revokes all of a user's privileges.

{ALTER | CREATE | DROP} ON *table_name*

Revokes the ability to alter, create, or drop tables and other database objects.

FILE

Revokes the ability to load data from or write data to files using the *SELECT INTO* and *LOAD DATA* commands.

INDEX

Revokes the ability to create or drop indexes.

PROCESS

Revokes the ability to view running processes using *SHOW FULL PROCESSLIST*.

RELOAD

Revokes the ability to invoke the *FLUSH* command.

SHUTDOWN

Revokes usage of the *mysqladmin shutdown* command to kill the server process.

CREATE TEMPORARY TABLES

Revokes the ability to create temporary tables.

LOCK TABLES

Revokes the use of the MySQL command *LOCK TABLES* on tables where the user has *SELECT* privileges.

REPLICATION CLIENT

Revokes the privilege to see metadata about replication slaves and masters.

REPLICATION SLAVE

Revokes the right to read binlogs of a replication master from a replication slave.

SHOW DATABASES

Revokes the privilege to execute the MySQL command *SHOW DATABASES*.

SUPER

Revokes the user privilege of opening one connection even if the *max_connections* threshold is reached. Also revokes from the users the privileges to execute these MySQL commands: *CHANGE MASTER*, *KILL*, *mysqladmin debug*, *PURGE [MASTER] LOGS*, and *SET GLOBAL*.

ON {*table_name* | * | *.* | *database_name*.*}

Removes privileges from the specified *table_name*, from all tables within the current database with *, from all tables in the specified *database_name* with *database_name*.*, or from all tables in all databases with *.*

FROM

Removes the privileges of one or more users, separated by commas. Usernames may also include an *@host_name* suffix, if you wish to restrict the revocation to a specific host computer.

REVOKE has size limitations. Usernames cannot be longer than 16 characters, while host, database, and database object names can be up to 60 characters in size. Usernames may also be linked to a specific host. Refer to the *GRANT* command for more details.



MySQL's implementation of *REVOKE* does not explicitly roll back permissions on objects that are dropped. Thus, it is necessary to explicitly *REVOKE* permissions on a table, even if the table is dropped. For example, let's say you drop a table without issuing a *REVOKE* on the existing permissions. If you later recreate the table, all of the old permissions will still be in place. Similarly, a user's privileges still persist in the database, even after the user has been dropped.

It is also important to note that MySQL allows multiple levels of permissions. Thus, a user might have access to a table from a table-level grant and have an additional set of permissions to the same table because they have been given global database or server-wide permissions. This means you need to be careful when revoking permissions because global level permissions might continue to provide permissions to a user when you thought you had revoked them!

The first command shown below revokes all privileges on the **sales** table for users **emily** and **dylan**, while the second command revokes select privileges for the user **kelly** in the current database:

```
REVOKE ALL PRIVILEGES ON sales FROM emily, dylan;
```

```
REVOKE SELECT ON * FROM kelly;
```

The first command below removes **kelly**'s ability to grant privileges to other users on the **sales** table, while the second command removes privileges that the user **sam** has in the **pubs** database:

```
REVOKE GRANT OPTION ON sales FROM kelly;
```

```
REVOKE ALL ON pubs.* FROM sam;
```

Oracle

The *REVOKE* command not only immediately revokes object and system privileges, it also revokes a role from a user (or a role from another role). Refer to the *GRANT* statement for more information on the specific object and system privileges supported by the *REVOKE* command.



The two forms of the *REVOKE* command, *REVOKE object_privilege* and *REVOKE system_privilege*, are mutually exclusive. Do not attempt to do both operations in a single statement. Since the complete syntax for both forms of the command are extremely long, refer to the [GRANT Statement](#) section for more details on the different *object_privileges* and *system_privileges*.

Oracle provides an interesting twist on privileges. While other database platforms typically allow a user to have more than one context for a set of privileges (based on the individual user and again for any groups that they are a member of), Oracle takes this one step further. A user may also have many grantors who have given them permission on a specific object. When this is the case, *ALL* grantors must revoke the privilege to effectively rid the user of the privilege. If even one grantor doesn't do this, then the user still has the privilege. Use the following syntax with *REVOKE*:

```
REVOKE { [object_privilege] [,...]  
        [system_privilege] |  
        [role] }  
[ON { [schema_name.][object] |  
      [DIRECTORY directory_object_name] |  
      [JAVA { [SOURCE | RESOURCE] } ] [schema_name.][object] } ]  
FROM {grantee_name [,...] | role_name [,...] | PUBLIC}
```


[CASCADE [CONSTRAINTS]] [FORCE];

where:

object_privilege

Revokes the specified privileges for a given user, *grantee_name*, or role against the named schema object, for example, a table or view.

ALL [PRIVILEGES]

Revokes all granted privileges on a named schema object. Since *ALL* also includes the *REFERENCES* privilege, you must include the *CASCADE* clause (see *REFERENCES* below).

ALTER

Revokes the privilege to change an existing table using the *ALTER TABLE* statement.

EXECUTE

Revokes the privilege to execute a stored procedure, user-defined function, or package.

INDEX

Revokes the privilege to create indexes on a table.

REFERENCES

Revokes the privilege to define referential integrity constraints. Requires that the *CASCADE CONSTRAINTS* clause also be used.

{ *SELECT* | *INSERT* | *DELETE* | *UPDATE* }

Revokes the privilege to execute each of the respective SQL commands against the named schema object. Note that *DELETE* privileges are dependent on *SELECT* privileges.

system_privilege

Revokes the specified Oracle system privilege from a given user, *grantee_name*, or role against the named system privilege such as *CREATE TRIGGER* or *ALTER USER*. Do not use the *ON* clause with the *REVOKE system_privileges* variation of the statement. Because there are so many and varied system privileges, please refer to the section under Oracle's implementation in the [GRANT Statement](#) section for a full list of system privileges.

role

Revokes a user from the named role or one role from another role.

ON

Revokes the named privilege for a given user or role from the named object. Objects include tables, views, sequences, stored procedures, user-defined functions, packages, materialized views, user-defined types, libraries, indextypes, user-defined operators, or a synonym for any of these. If you do not include the schema name, Oracle assumes the schema of the current user. For example, you could revoke *SELECT* privileges for a given user on the **scott.authors** table. Oracle also supports two additional keywords for special cases:

DIRECTORY *directory_object_name*

Identifies a directory object from which the privileges are revoked.

```
JAVA [ { SOURCE | RESOURCE } ] [schema_name.][object ] }
```

Identifies a Java source or a resource schema object from which the privileges are revoked.

```
FROM {grantee_name | role_name | PUBLIC}
```

Names the user or role who will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role. Multiple grantees can be listed with a comma between each.

```
CASCADE [CONSTRAINTS]
```

Drops any referential integrity constraints that depend on the privilege being revoked in the first place. This clause is needed only when the *REFERENCES* or *ALL [PRIVILEGES]* clauses are used.

```
[FORCE]
```

Required to revoke *EXECUTE* object privileges on user-defined type objects with type or table dependencies.

Oracle automatically cascades the revocation from the defined *grantee_name* to all users that received their privileges from the grantee. In addition, any objects created by the grantee that depend on the privilege (such as stored procedures, triggers, views, and packages that depend on a *SELECT* privilege against a certain table) will become invalid.

Users who are granted the *GRANT ANY ROLE* system privilege may revoke any role. The *REVOKE* command can only revoke privileges specifically granted with the *GRANT* command, not privileges available through roles or the operating system. In a case like this, you must use the *REVOKE* command to drop the privilege from the role. All users assigned to the role will then lose the privilege.

The following is an example of revoking a role from specific grantee and revoking a system privilege from a role:

```
REVOKE read-only FROM sarah;  
  
REVOKE CREATE ANY SEQUENCE,  
CREATE ANY DIRECTORY FROM read_only;
```

Here's an example that revokes a *REFERENCES* privilege and cascades the revoked privileges:

```
REVOKE REFERENCES  
ON pubs_new_york.emp  
FROM dylan  
CASCADE CONSTRAINTS;
```

The following example grants all privileges on a specific table, and then revokes a privilege:

```
GRANT ALL PRIVILEGES ON emp TO dylan;  
  
REVOKE DELETE, UPDATE ON emp FROM dylan;
```

PostgreSQL

PostgreSQL offers base-level features of the *REVOKE* statement, primarily for revoking privileges on specific tables, views, and sequences. It does not offer the ANSI standard special options like *HIERARCHY OPTION FOR* or *ADMIN OPTION FOR*. The syntax is as follows:

```
REVOKE { [ GRANT OPTION FOR ] |  
privileges [...] }
```

```
ON {object_name} [,...]  
FROM {grantee_name [,...]} | PUBLIC | GROUP group_name}  
[ {CASCADE | RESTRICT} ]
```

where:

GRANT OPTION FOR

Revokes the privilege of a user to grant privileges to other users.

privileges

Revokes privileges for a variety of statements, which may be combined in any order:

ALL PRIVILEGES

A shorthand for every privilege the grantor has available to grant. Revokes all privileges currently assigned to the named users and/or for the specified database objects. Generally not a recommended approach, since it can encourage sloppy coding.

```
{SELECT | INSERT | DELETE | UPDATE}
```

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

REFERENCES

Revokes the user's privilege to create or drop foreign key constraints that reference the database object as a parent object.

RULE

Revokes the user's privilege to create or drop a rule on a table or view.

USAGE

Revokes privileges to use a domain, user-defined type, or character set, as opposed to the *UNDER* privilege, which controls the ability to create these objects.

```
FROM grantee_name [,...]} | PUBLIC | GROUP group_name
```

Names the user or role who will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role (a role that implicitly includes all users). Multiple grantees can be listed with a comma between each.

CASCADE | RESTRICT

Limits a revoke operation to only the specified privilege (*RESTRICT*) or causes the specified privilege and all dependent privileges to be revoked (*CASCADE*). Only used with the *GRANT OPTION FOR* clause.

PostgreSQL's implementation of *REVOKE* is relatively straightforward. The only issue to be aware of is that PostgreSQL treats the term *GROUP* as a synonym for *ROLE*. For example, the following removes some privileges from the *PUBLIC* group and from the *READ-ONLY* group:

```
REVOKE ALL PRIVILEGES ON employee FROM public;
```

```
REVOKE SELECT ON jobs FROM read-only;
```

PostgreSQL *does not* support privileges on individual columns within a table or view. In order to work around this limitation, you can create a view with only the columns in question and *GRANT* or *REVOKE* privileges to and from that view.

When revoking the *GRANT OPTION FOR* clause, you should pay extra attention to dependencies. If you revoke a user's privilege with the *RESTRICT* keyword, the statement will fail if other users depend on the revokee. If you revoke a user's privilege with the *CASCADE* keyword, the statement will revoke privileges not only for the revokee, but for all users that depend on the revokee.

SQL Server

SQL Server implements the *REVOKE* statement as a means to undo any permission settings for a given user. This is significant because SQL Server supports an extension statement called *DENY* which explicitly disallows a user from a specific resource. *REVOKE*, in SQL Server, can be used to undo permissions granted to a user with *GRANT*. If you want to explicitly prevent a user from having a certain privilege, you must use the *DENY* statement.

SQL Server does not support the *HIERARCHY OPTION* and *ADMIN OPTION* clauses of the ANSI standard. Although the *ADMIN OPTION* clause is not supported, a couple of administrative privileges (*CREATE* and *BACKUP*) are supported in SQL Server's version of *REVOKE*. Use the following syntax:

```
REVOKE [GRANT OPTION FOR]
{ [object_privilege] [,...]}
  [system_privilege ]
[ON [object] [(column [,...])] ]
{TO | FROM} {grantee_name [,...] | role [,...] | PUBLIC | GUEST }
[CASCADE]
[AS {group_name | role_name} ]
```

where:

GRANT OPTION FOR

Revokes the privilege of a user to grant a specific privilege to other users.

object_privilege

Revokes privileges for a variety of statements, which may be combined in any order:

ALL PRIVILEGES

Revokes all privileges currently assigned to the named users and/or for the specified database objects. We do not recommend this approach, since it can encourage sloppy programming.

{*SELECT* | *INSERT* | *DELETE* | *UPDATE*}

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

REFERENCES

Revokes the user's privilege to create or drop foreign key constraints that reference the database object as a parent object.

RULE

Revokes the user's privilege to create or drop a rule on a table or view.

EXECUTE

Revokes privileges to execute a stored procedure, user-defined function, or extended stored procedure.

system_privilege

Revokes the privilege to execute any of the following statements: *CREATE DATABASE*, *CREATE DEFAULT*, *CREATE FUNCTION*, *CREATE PROCEDURE*, *CREATE RULE*, *CREATE TABLE*, *CREATE VIEW*, *BACKUP DATABASE*, and *BACKUP LOG*.

ON *object* [(*column* [...])]

Removes the privilege for the user on the named object. If the object is a table or view, you may optionally revoke privileges on specific columns. You may revoke *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *REFERENCES* privileges on a table or view. You may revoke only *SELECT* and *UPDATE* privileges on the columns of a table or view. You may revoke *EXECUTE* privileges on a stored procedure, user-defined function, or extended stored procedure.

{TO | FROM} *grantee_name* | *role* | PUBLIC | GUEST

Names the user or role who will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role (a role that implicitly includes all users). Multiple grantees can be listed with a comma between each. SQL Server also supports the role known as *GUEST*, an account used by all users who do not have a user account in the database.

CASCADE

Revokes privileges on users who received their permissions under the *WITH GRANT OPTION* clause. Required when using the *GRANT OPTION FOR* clause.

AS {*group_name* | *role_name*}

Defines the authority by which the privilege is revoked. In certain circumstances, a user may need to temporarily assume the privileges of a group to revoke the specified privileges. To accommodate this scenario, you can use the *AS* clause to assume the authority of a given group.

The two forms of the *REVOKE* command, *REVOKE object_privilege* and *REVOKE system_privilege*, are mutually exclusive. Do not attempt to do both operations in a single statement. The key syntactical difference between them is that you should not include the *ON* clause when revoking system privileges. For example, to drop a system privilege:

```
REVOKE CREATE DATABASE, BACKUP DATABASE FROM dylan, katie
```

If commands were granted to a user *WITH GRANT OPTION* enabled, the privilege should be revoked using both *WITH GRANT OPTION* and *CASCADE* together. For example:

```
REVOKE GRANT OPTION FOR
```

```
SELECT, INSERT, UPDATE, DELETE ON titles
```

```
TO editors
```

```
CASCADE
```

```
GO
```

REVOKE can only be used in the current database. Consequently, the *CURRENT_USER* and *CURRENT_ROLE* options in the ANSI variations are always implied. *REVOKE* also is used to disable any *DENY* settings.



SQL Server additionally supports the *DENY* statement. *DENY* is syntactically the same as *REVOKE*. However, it is conceptually different in that *REVOKE* neutralizes a user's privileges, while *DENY* explicitly prohibits a user's privileges. Use the *DENY* statement to keep a user or role from accessing a privilege, even a privilege granted through a role or an explicit permission.

You must use *REVOKE* to remove previously granted *OR* denied privileges. For example, the user **kelly** took an extended leave of absence for maternity leave. During that time, her permissions on the employee table where denied. Now that she's returned, we'll lift the denied privileges:

```
DENY ALL ON employee TO Kelly
```

```
GO
```

```
REVOKE ALL ON employee TO Kelly
```

```
GO
```

In this example, the *REVOKE* command did not remove her privileges; rather, it neutralized the *DENY* statement.

See Also

GRANT

ROLLBACK Statement

The *ROLLBACK* statement undoes a transaction to its beginning or to a previously declared *SAVEPOINT*. *ROLLBACK* also closes any open cursors.

Platform	Command
DB2	Supported
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with variations

SQL2003 Syntax

```
ROLLBACK [WORK]
```

```
[AND [NO] CHAIN]
```

```
[TO SAVEPOINT savepoint_name]
```

Keywords

WORK

An optional keyword, but basically just noise.

AND [NO] CHAIN

AND CHAIN tells the DBMS to end the current transaction, but to share the common transaction environment (such as transaction isolation level) with the next transaction. *AND NO CHAIN* simply ends the transaction (and is effectively the same as not including the clause at all).

TO SAVEPOINT *savepoint_name*

Allows the transaction to be rolled back to a named savepoint (that is, a partial rollback) rather than rolling back the entire transaction. The *savepoint_name* may be a literal expression or a variable. If no savepoint of *savepoint_name* is active, the statement will return an error. When the *TO SAVEPOINT* clause is omitted, all cursors are closed. When the *TO SAVEPOINT* clause is included, only the cursors that were open within the savepoint are closed.

In addition to undoing a single data-manipulation operation such as an *INSERT*, *UPDATE*, or *DELETE* statement (or a batch of them), the *ROLLBACK* statement undoes transactions up to the last issued *START TRANSACTION*, *SET TRANSACTION*, or *SAVEPOINT* statement.

Rules at a Glance

ROLLBACK is used to undo a transaction. It can be used to undo explicitly declared transactions that are started with a *START TRAN* statement or implicitly with a transaction-initiating statement. It can also be used to undo implicit transactions that are started without a *START TRAN* statement. *ROLLBACK* is mutually exclusive of the *COMMIT* statement.

Most people associate commands like *INSERT*, *UPDATE*, and *DELETE* with the term "transaction." However, transactions encompass a wide variety of commands. This list of commands varies from platform to platform, but generally includes any command that alters data or database structures and is logged to the database logging mechanism. According to the ANSI standard, all SQL statements can be undone with *ROLLBACK*.

Programming Tips and Gotchas

The most important gotcha to consider is that some database platforms perform automatic and *implicit* transactions, while others require *explicit* transactions. If you assume a platform uses one method of transactions over the other, you may get bitten. Thus, when moving between database platforms, you should follow a standard, preset way of addressing transactions. We recommend staying with explicit transactions using *SET TRAN* or *START TRAN* to begin a transaction, and *COMMIT* or *ROLLBACK* to end a transaction.

DB2

DB2 supports the basic ANSI form of the statement. It does not support the *[AND [NO] CHAIN]* clause. (Also note that DB2 does not support the *START TRANSACTION* statement, so all transactions in DB2 start implicitly.) DB2's *ROLLBACK* syntax is as follows:

```
ROLLBACK [WORK] [TO SAVEPOINT savepoint_name]
```

A number of DB2 *SET* statements are not covered by transaction control and, thus, cannot be rolled back. These include: *SET CONNECTION*, *SET CURRENT DEFAULT TRANSFORM GROUP*, *SET CURRENT DEGREE*, *SET CURRENT EXPLAIN MODE*, *SET CURRENT EXPLAIN SNAPSHOT*, *SET CURRENT PACKAGESET*, *SET CURRENT QUERY OPTIMIZATION*, *SET CURRENT REFRESH AGE*, *SET EVENT MONITOR STATE*, *SET PASSTHRU*, *SET PATH*, *SET SCHEMA*, and *SET SERVER OPTION*.

DB2 issues an implicit rollback any time a unit of work terminates abnormally.

When a regular (non-savepoint) rollback occurs on DB2, all locks held by the unit of work are released, open cursors are closed, *LOB* locators are freed, and in some cases, caching is impacted. When a rollback to savepoint occurs on DB2, locks and *LOB* locators are preserved, and dynamically prepared SQL is still valid (though it will have to be parsed again). Rollback to savepoint involving cursors is a bit more complicated. If a cursor depends on DDL in the savepoint, it will be marked invalid. If a cursor is referenced in the savepoint but does not contain DDL used by the savepoint, the cursor will remain open and the cursor pointer will be positioned before the next row. In all other cases, the cursor remains unaffected.

MySQL

MySQL supports only the simplest rollback mechanism and, even then, you must declare any table you might issue a *ROLLBACK* against to be *transaction safe*. (A transaction safe table is one declared with the InnoDB or BDB property. Refer to *CREATE TABLE* for more information.)

```
ROLLBACK [TO SAVEPOINT savepoint_name]
```

MySQL allows you to issue transaction control statements like *COMMIT* or *ROLLBACK* against non-transaction safe tables, but it will simply ignore them and autocommit as usual. In the case of *ROLLBACK* against a non-transaction safe table, changes will not be rolled back.

MySQL, by default, runs in *AUTO COMMIT* mode, causing all data modifications to automatically be written to disk. You can set *AUTO COMMIT* off by issuing the command *SET AUTO COMMIT=0*. You can also control the autocommit behavior on a statement-by-statement basis using the *BEGIN* or *BEGIN WORK* command:

```
BEGIN;  
  
SELECT @A:=SUM(salary) FROM employee WHERE job_type=1;  
  
BEGIN WORK;  
  
UPDATE jobs SET summary=@A WHERE job_type=1;  
  
COMMIT;
```

MySQL automatically issues an implicit *COMMIT* upon the completion of any of these statements: *ALTER TABLE*, *BEGIN*, *CREATE INDEX*, *DROP DATABASE*, *DROP TABLE*, *RENAME TABLE*, and *TRUNCATE*.

MySQL supports partial rollbacks using savepoints with Version 4.0.14.

Oracle

Oracle supports the ANSI standard form of the *ROLLBACK* statement with the addition of the *FORCE* clause:

```
ROLLBACK [WORK] {[TO [SAVEPOINT] savepoint_name] |  
  
[FORCE 'text']};
```

ROLLBACK clears all data modifications made to the current open transaction (or to a specific, existing savepoint). It also releases all locks held by the transaction, erases all savepoints, undoes all the changes of the current transaction, and ends the current transaction.

ROLLBACK ... TO SAVEPOINT rolls back just that portion of the transaction after the savepoint, erases all savepoints that followed, and releases all table- and row-level locks taken after the savepoint. Refer to the [SAVEPOINT Statement](#) section for more information.

Oracle's implementation closely follows the SQL standard with the exception of the *FORCE* option. *ROLLBACK FORCE* rolls back an in-doubt, distributed transaction. You must have the *FORCE TRANSACTION* privilege to issue a *ROLLBACK...FORCE* statement. *FORCE* cannot be used with *TO [SAVEPOINT]*. *ROLLBACK...FORCE* affects the transaction named in *'text'* and not the current transaction, where *'text'* must be equal to the local or global transaction ID of the transaction you want to rollback. (These transactions and their ID names are detailed in the Oracle system view **DBA_2PC_PENDING**.)

For example, you might want to roll back your current transaction to the **salary_adjustment** savepoint. These two commands are equivalent:

```
ROLLBACK WORK TO SAVEPOINT salary_adjustment;  
  
ROLLBACK TO salary_adjustment;
```

In the following example, you roll back an in-doubt distributed transaction:

```
ROLLBACK FORCE '45.52.67'
```

PostgreSQL

PostgreSQL supports the basic form of *ROLLBACK*, but it does not support savepoints.

```
{ROLLBACK | ABORT} [WORK | TRANSACTION];
```

ROLLBACK clears all data modifications made to the current open transaction. It will return an error if no transaction is currently open. PostgreSQL supports both the *WORK* option and the *TRANSACTION* option. It does not support rolling back to a savepoint. PostgreSQL supports *ABORT* as a synonym of *ROLLBACK*.

SQL Server

SQL Server supports both the *WORK* and *TRAN* keywords. The only difference between them is that the *ROLLBACK WORK* statement doesn't allow rolling back of a named transaction or to a specific savepoint.

```
ROLLBACK { [WORK] | [TRAN[SACTION]  
  [transaction_name | savepoint_name] ] }
```

If *ROLLBACK* is issued alone without the *WORK* or *TRAN* keywords, it rolls back all current open transactions. *ROLLBACK* normally frees locks, but it does not free locks when rolling back to a savepoint.

SQL Server allows you to name a specific *transaction_name* in addition to a specific *savepoint_name*. You may reference them explicitly, or you may use variables within Transact-SQL.

ROLLBACK TRANSACTION, when issued in a trigger, undoes all data modifications, including those performed by the trigger, up to the point of the *ROLLBACK* statement. Nested triggers are not executed if they follow a *ROLLBACK* within a trigger; however, any statements within the trigger that follow the rollback are not impacted by the rollback. *ROLLBACK* behaves similarly to *COMMIT* with regard to nesting, resetting the *@@TRANCOUNT* system variable to zero. (Refer to the [COMMIT Statement](#) section for more information on transaction control within a SQL Server nested trigger.)

Following is a Transact-SQL batch using *COMMIT* and *ROLLBACK* in Microsoft SQL Server. In the example, the code inserts a record into the **sales** table. If the insertion fails, the transaction is rolled back; if the insertion succeeds, the transaction is committed:

```
BEGIN TRAN -- initializes a transaction  
  
-- the transaction itself  
  
INSERT INTO sales  
  
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832')  
  
-- some error-handling in the event of a failure  
  
IF @@ERROR <> 0  
  
BEGIN  
  -- raises an error in the event log and skips to the end  
  RAISERROR 50000 'Insert of sales record failed'  
  
  ROLLBACK WORK  
  
  GOTO end_of_batch  
  
END  
  
-- the transaction is committed if no errors are detected  
  
COMMIT TRAN  
  
-- the GOTO label that enables the batch to skip to  
  
-- the end without committing  
  
end_of_batch:  
  
GO  
  
SAVEPOINT sales1
```

See Also

```
COMMIT  
RELEASE SAVEPOINT  
SAVEPOINT
```

SAVEPOINT Statement

This command breaks a transaction into logical breakpoints. Multiple savepoints may be specified within a single transaction. The main benefit of the *SAVEPOINT* command is that transactions may be partially rolled back to a savepoint marker using the *ROLLBACK* command.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

SAVEPOINT savepoint_name

Keywords

SAVEPOINT savepoint_name

Establishes a savepoint named *savepoint_name* within the current transaction.

Some vendors allow duplicate savepoint names within a transaction, but this is not recommended by the ANSI standard.

SQL2003 supports the statement *RELEASE SAVEPOINT savepoint_name*, enabling an existing savepoint to be eliminated. Refer to the [RELEASE SAVEPOINT Statement](#) section for more information about eliminating an existing savepoint.

Rules at a Glance

Savepoints are established within the scope of the entire transaction in which they are defined. Savepoint names should be unique within their scope. Furthermore, make sure you use *BEGIN* and *COMMIT* statements prudently, because accidentally placing a *BEGIN* statement too early or a *COMMIT* statement too late can have a dramatic impact on the way transactions are written to the database. Always make sure to provide easy-to-understand names for your savepoints because you'll be referencing them later in your programs.

Programming Tips and Gotchas

Generally, reusing a savepoint name won't produce an error or warning. A duplicate savepoint name will render the previous savepoint with the same name useless. So be careful when naming savepoints!

When a transaction is initiated, resources (namely, locks) are expended to ensure transactional consistency. Make sure that your transaction runs to completion as quickly as possible so that the locks are released for others to use.

DB2

DB2 supports a more elaborate version of *SAVEPOINT* than is specified in the ANSI standard.

SAVEPOINT savepoint_name [UNIQUE]

ON ROLLBACK RETAIN CURSORS [ON ROLLBACK RETAIN LOCKS]

where:

SAVEPOINT savepoint_name

Establishes a savepoint named *savepoint_name* within the current transaction.

UNIQUE

Indicates that the savepoint is intended to be unique and that the savepoint name cannot be reused or overwritten. (Supported only on DB2 for OS/390.)

ON ROLLBACK RETAIN CURSORS

Specifies that open cursors remain open when rolling back to the savepoint. (More details are given in the [ROLLBACK Statement](#) section.)

ON ROLLBACK RETAIN LOCKS

Specifies that locks are retained when rolling back to the named savepoint. This behavior is the default when the clause is omitted. (More details are given in the [ROLLBACK Statement](#) section.)

An important note about DB2 is that savepoints cannot be nested. An error will occur any time you issue a savepoint where an active savepoint already exists.

DB2 implicitly releases any savepoints within the transaction if it is committed.

In the following example, we set the **sales_processing** savepoint without explicitly retaining locks:

```
SAVEPOINT sales_processing ON ROLLBACK RETAIN CURSORS;
```

However, since retaining locks is the default, the next example is essentially redundant:

```
SAVEPOINT sales_processing ON ROLLBACK RETAIN  
CURSORS ON ROLLBACK RETAIN LOCKS;
```

MySQL

Not supported.

Oracle

Oracle fully supports the ANSI implementation.

```
SAVEPOINT savepoint_name
```

The following example performs several data modifications, and then rolls back to a savepoint:

```
INSERT INTO sales  
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832');  
SAVEPOINT after_insert;  
UPDATE sales SET terms = 'Net 90'  
WHERE sales_id = '7896';  
SAVEPOINT after_update;  
DELETE sales;  
ROLLBACK TO after_insert;
```

PostgreSQL

Not supported.

SQL Server

SQL Server does not support the *SAVEPOINT* command. Instead, it uses the *SAVE* command:

SAVE TRAN[*SACTION*] *savepoint_name*

In addition, rather than declaring the literal name of the savepoint, you can optionally reference a variable containing the name of the savepoint. If you use a variable, it must be of the *CHAR*, *VARCHAR*, *NCHAR*, or *NVARCHAR* datatype.

SQL Server allows you to have many different named savepoints in a single transaction. However, be careful. Since SQL Server supports multiple savepoints in a single transaction, it might appear that SQL Server fully supports nested savepoints. In fact, it *does not*. Any time you issue a commit or savepoint in SQL Server, it only commits or rolls back to the last open savepoint.

When the *ROLLBACK TRAN savepoint_name* command is executed, SQL Server rolls the transaction back to the specified savepoint, then continues processing with the next valid Transact-SQL command following the *ROLLBACK* statement. Finally, the transaction must ultimately be concluded with a *COMMIT* or a final *ROLLBACK* statement.

See Also

COMMIT
ROLLBACK

SELECT Statement

The *SELECT* statement retrieves rows, columns, and derived values from one or many tables of a database.

Platform	Command
DB2	Supported, with variations (ANSI joins supported)
MySQL	Supported, with variations (ANSI joins partially supported)
Oracle	Supported, with variations (ANSI joins supported)
PostgreSQL	Supported, with variations (ANSI joins partially supported)
SQL Server	Supported, with variations (ANSI joins supported)

SQL2003 Syntax

The full syntax of the *SELECT* statement is powerful and complex, but can be broken down into these main clauses:

SELECT [{*ALL* | *DISTINCT*}] *select_item* [*AS alias*] [...]

FROM [*ONLY* | *OUTER*]

 {*table_name* [[*AS alias*] | *view_name* [[*AS alias*]]} [...]

[[*join_type*] *JOIN join_condition*]

[*WHERE search_condition*] [{*AND* | *OR* | *NOT*} *search_condition* [...]]

```
[GROUP BY group_by_expression{group_by_columns | ROLLUP group_by_columns |  
    CUBE group_by_columns | GROUPING SETS ( grouping_set_list ) |  
    ( ) | grouping_set , grouping_set_list}  
[HAVING search_condition] ]  
[ORDER BY {order_expression [ASC | DESC]} [,...]]
```

Keywords

Each of the keywords shown below, except the *select_item* clause, is discussed in greater detail in the "Rules at a Glance" section that follows:

[ALL | DISTINCT] *select_item*

Retrieves values that compose the query result set. Each *select_item* may be a literal, an aggregate or scalar function, a mathematic calculation, a parameter or variable, or a subquery, but a *select_item* is most commonly a column from a table or view. A comma must separate each item in a list of such items.

The schema or owner name should be prefixed to a column when extracted from a context outside of the current user. If the table is owned by another user, then that user must be included in the column reference. For example, assume the user **jake** needs to access data in the schema **katie**:

```
SELECT emp_id  
FROM katie.employee;
```

You can use the asterisk (*) shorthand to retrieve all columns in every table or view shown in the *FROM* clause. It's a good idea to use this shortcut on single-table queries only.

ALL, the default behavior, returns all records that meet the selection criteria. *DISTINCT* tells the database to filter out any duplicate records, thus retrieving only one instance out of many identical records.

AS *alias*

Replaces a column heading (when in the *select_item* clause) or a table name or view name (when in the *FROM* clause) with a shorter heading or name. This clause is especially useful for replacing cryptic or lengthy names with short, easy to understand names or mnemonics, or when the column is only derived data so you don't end up with a column called **ORA000189x7/0.02**. It is also very useful in self-joins and correlated subqueries where a single query references the same table more than once. When multiple items appear in the *select_item* clause or *FROM* clause, make sure to place the commas *after* an *AS alias* clauses. Be careful to always use an alias uniformly once you introduce it into the query.

FROM *table_name*

Serves to list all tables and/or views from which the query retrieves data. Separate table and view names using commas. The *FROM* clause also allows you to assign aliases to long table/or view names or subqueries using the *AS* clause. Using shorter aliases instead of longer table or view names simplifies coding. (Of course, this might thwart the DBA's carefully planned naming conventions, but the alias only lasts for the duration of the query. Refer to the following section on "Rules at a Glance.") *FROM* clauses may contain a subquery (refer to the [SUBQUERY Substatement](#) section).

ONLY

Specifies that only the rows of the named table or view, and no rows in subtables or subviews, will be retrieved in the result set. When using *ONLY*, be sure to enclose the *table_name* or *view_name* within parentheses. *ONLY* is ignored if the table or view has no subtables or subviews.

OUTER

Specifies that the rows of the named table or view, along with the rows and columns of any and all subtables or subviews, will be retrieved in the result set. Columns of the subtables (or subviews) will be appended to the right, in subtable hierarchy order according to depth of the subtable. In extensive hierarchies, subtables with common parents are appended in creation order of their types. When using *OUTER*, be sure to enclose the *table_name* or *view_name* within parentheses. *OUTER* is ignored if the table or view has no subtables or subviews.

JOIN *join_condition*

Joins together the result set of the table shown in the *FROM* clause to another table that shares a meaningful relationship based on a common set of values. These values are usually contained in columns of the same name and datatype that appear in both tables being joined. These columns, or possibly a single column from each table, are called the *join key* or *common key*. Most—but not all—of the time, the join key is the primary key of one table and a foreign key in another table. As long as the data in the columns match, the join can be performed. (Note that joins can also be performed using the *WHERE* clause. This technique is sometimes called a *theta join*.)

Join_conditions are most commonly depicted in the form:

```
JOIN table_name2 ON table_name1.column1 <comparison operator>
```

```
table_name2.column2
```

```
JOIN table_name3 ON table_name1.columnA <comparison operator>
```

```
table_name3.columnA
```

```
[...]
```

When the *comparison_operator* is the equal sign (=), a join is said to be an *equijoin*. However, the comparison operation could be any acceptable comparison operation like <, >, <=, >=, or even <>.

Use the *AND* operator to issue a *JOIN* with multiple conditions. You can also use the *OR* operator to specify alternate join conditions.

If an explicit *join_type* is omitted, then an *INNER JOIN* is assumed. Note that there are many types of joins, each with their own rules and behaviors, described in the "Rules at a Glance" (and in more detail in the [JOIN Subclause](#) section). Note that an alternate approach to the join condition, via the *USING* clause, exists:

```
USING (column_name [,...])
```

Acts as an alternative to the *ON* clause. Instead of describing the conditions of the join, simply provide a *column_name* (or column names separated by commas) that appears in both tables. The database then evaluates the join based on the column (or columns) of *column_name* that appear in both tables. (The column names must be identical in both tables.) In the following example, the two queries produce identical results:

```
SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);
```

```
SELECT emp_id
FROM employee AS e
LEFT JOIN sales AS s
  ON e.emp_id = s.emp_id
  AND e.region_id = s.region_id;
```

WHERE *search_condition*

Filters unwanted data from the result set of the query, returning only those records that satisfy the search conditions. A poorly written *WHERE* clause can ruin the performance of an otherwise useful *SELECT* statement, so the nuances of the *WHERE* clause *must be mastered* thoroughly. *Search_conditions* are syntactically depicted in the form:

WHERE [*schema*.*table_name*.]*column operator value*

WHERE clauses usually compare the values contained in a *column* of the table. The values of the column are compared using an *operator* of some type (refer to [Chapter 2](#) for more detail). For example, a column might equal (=) a given *value*, be greater than (>) a given *value*, or be *BETWEEN* a range of *values*.

WHERE clauses may contain many search conditions concatenated together using the *AND* or *OR* Boolean operators. Parentheses can be used to impact the order of precedence of the search conditions. *WHERE* clauses can contain subqueries (refer to the [SUBQUERY Substatement](#) section).

GROUP BY *group_by_expression*

Used in queries that utilize aggregate functions such as *AVG*, *COUNT*, *COUNT DISTINCT*, *MAX*, *MIN*, and *SUM* to group result sets into the categories you define in the *group_by_expression*. The *group_by_expression* of the *GROUP BY* clause has an elaborate syntax of its own:

[*GROUP BY* *group_by_expression*

where *group_by_expression* is:

```
{ (grouping_column [...]) | ROLLUP (grouping_column [...]) |  
  CUBE (grouping_column [...]) | GROUPING SETS ( grouping_set_list ) |  
  ( ) | grouping_set , grouping_set_list}
```

Refer below to the subsection [The GROUP BY clause](#) in the "Rules at a Glance" section for examples and more information on *ROLLUP*, *CUBE*, and *GROUPING SETS*.

HAVING *search_condition*

Adds search conditions on the results of the *GROUP BY* clause in a manner similar to the *WHERE* clause. *HAVING* does not affect the rows used to calculate the aggregates. *HAVING* clauses may contain a subquery. (Refer to the [SUBQUERY Substatement](#) section for more information).

ORDER BY *order_expression* [ASC | DESC]

Sorts the result set in either ascending (ASC) or descending (DESC) order using the *order_expression* specified. The *order_expression* is a list of columns, separated by commas, that you want to sort by.

Rules at a Glance

Each clause of the *SELECT* statement has a specific use. Thus, it is possible to speak individually of the *FROM* clause, the *WHERE* clause, the *GROUP BY* clause, and so forth. You can get more details and examples of *SELECT* statements by looking up the entries for each clause of the statement. However, not every query needs every clause. At a minimum, a query needs a *SELECT* item list and a *FROM* clause. Because the *SELECT* clause is so important and offers so many options, we've divided "Rules at a Glance" into the following detailed subsections:

- Aliases and *WHERE* clause joins
- The *JOIN* clause
- The *WHERE* clause
- The *GROUP BY* clause
- The *HAVING* clause
- The *ORDER BY* clause

Aliases and *WHERE* clause joins

Columns may need to be prefixed with the database, schema, and table name, particularly when there may be the same column name in one or more tables of the query. For example, (on an Oracle database) both the **jobs** table and **scott's employee** table contain the **job_id** column. Note that the following examples join the **employee** and **jobs** table using the *WHERE* clause:

```
SELECT  scott.employee.emp_id,
        scott.employee.fname,
        scott.employee.lname,
        jobs.job_desc
FROM    scott.employee,
        jobs
WHERE   scott.employee.job_id = jobs.job_id
ORDER BY scott.employee.fname,
        scott.employee.lname
```

You can also use aliases to write a query more simply and easily:

```
SELECT  e.emp_id,
        e.fname,
        e.lname,
        j.job_desc
FROM    scott.employee AS e,
        jobs AS j
WHERE   e.job_id = j.job_id
ORDER BY e.fname,
        e.lname
```

These two queries also illustrate some important rules about *WHERE* clause joins:

1. Use commas to separate multiple elements in a *select_item* list, tables in the *FROM* clause, and items in the *order_expression*.
2. Use the *AS* clause to define aliases.
3. Use aliases consistently throughout the *SELECT* statement once you define them.

In general, you should favor the *JOIN* clause (explained in a moment) over the *WHERE* clause for describing join expressions. This not only keeps your code cleaner, making it easy to differentiate join conditions from search conditions, it also allows you to avoid the possibility of counter-intuitive behavior that results from using the *WHERE* clause for outer joins in some implementations.

The JOIN clause

To perform the same query as in the previous example using an ANSI-style join, list the first table and the keyword *JOIN*, followed by the table to be joined. Once the second table is typed in, type the keyword *ON* and the join condition that would have been used in the old style query. The next example shows the query now in ANSI style.

```
SELECT  e.emp_id, e.fname, e.lname, j.job_desc
FROM    scott.employee AS e
JOIN    jobs AS j ON e.job_id = j.job_id
ORDER BY e.fname, e.lname;
```


Alternately, you could use the *USING* clause. Instead of describing the conditions of the join, simply provide one or more *column_names* (separated by commas) that appears in both of the joined tables. The database then evaluates the join based on the column (or columns) of *column_name* that appear in both tables. (The column names must be identical in both tables.) In the following example, the two queries produce identical results, one using the *ON* clause and one using the *USING* clause:

```
SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);

SELECT emp_id
FROM employee AS e
LEFT JOIN sales AS s
  ON e.emp_id = s.emp_id
  AND e.region_id = s.region_id;
```

You can specify several different types of joins in ANSI style:

Cross join

Specifies the complete cross product of two tables. For each record in the first table, all the records in the second table are joined, creating a potentially *huge* result set. (Of course, the result set would be small if both tables had only four records each, but imagine if they had four million!) This command has the same effect as leaving off the join condition and is also known as a "Cartesian Product." We do not recommend using cross joins.

Inner join

Specifies that unmatched rows in either table of the join should be discarded. If no join type is explicitly defined in the ANSI style, then this is the default.

Left [outer] join

Specifies that all records be returned from the table on the left side of the join statement. If a record returned from the left table has no matching record in the table on the right side of the join, it is still returned. Columns from the right table return NULL values. Many professionals recommend configuring outer joins as left joins wherever possible for consistency.

Right [outer] join

Specifies that all records be returned from the table on the right side of the join statement, even if the table on the left has no matching record. Columns from the left table return NULL values.

Full [outer] join

Specifies that all rows from either table be returned, regardless of matching records in the other table. The result set shows NULL values where no matching data exists in the join.




Not all joins are supported by all platforms, so refer to the [JOIN Subclause](#) section for full details on platform-specific join support.

The WHERE clause

A poorly written *WHERE* clause can ruin an otherwise beautiful *SELECT* statement, so the nuances of the *WHERE* clause *must be mastered* thoroughly. This is an example of a typical query and a multipart *WHERE* clause:

```
SELECT  a.au_lname,  
        a.au_fname,  
        t2.title,  
        t2.pubdate  
FROM    authors a  
JOIN    titleauthor t1 ON a.au_id = t1.au_id  
JOIN    titles t2 ON t1.title_id = t2.title_id  
WHERE   (t2.type = 'business' OR t2.type = 'popular_comp')  
        AND  t2.advance > 5500  
ORDER BY t2.title
```

In examining this query, the parentheses impact the order of processing for the search conditions. Use parentheses to move search conditions up or down in precedence just like you would in an algebra equation.

	On some platforms, the database's default collation (also known as <i>sort order</i>) impacts how the <i>WHERE</i> clause filters results for a query. For example, SQL Server is (by default) <i>dictionary-order</i> and <i>case-insensitive</i> , making no differentiation between "Smith," "smith," and "SMITH." But Oracle uses <i>dictionary-order</i> and is <i>case-sensitive</i> , finding the values "Smith," "smith," and "SMITH" to be unequal.
--	---

The *WHERE* clause offers many more specific capabilities than the preceding example illustrates. The list below references some of the more common capabilities of the *WHERE* clause. Refer to the [WHERE Clause](#) section for full details and examples.

NOT

Inverts a comparison operation using the syntax *WHERE NOT expression*. Thus, you might use *WHERE NOT LIKE...* or *WHERE NOT IN...* in a query.

Comparison operators

Compare any set of values using the operations *<*, *>*, *<>*, *>=*, *<=*, and *=*. For example:

```
WHERE emp_id = '54123'
```

IS NULL or IS NOT NULL conditions

Search for any NULL or NOT NULL values using the syntax *WHERE expression IS [NOT] NULL*.

AND

Merges multiple conditions, returning only those records that meet all conditions, using the *AND* operator. The maximum number of multiple conditions is determined by each platform. For example:

```
WHERE job_id = '12' AND job_status = 'active'
```

OR

Merges alternative conditions, returning records that meet any of the conditions, using the *OR* operator. For example:

```
WHERE job_id = '13' OR job_status = 'active'
```

LIKE

Tells the query to use a pattern-matching string contained within quotation marks. The wildcard symbols supported by each platform are detailed in the [LIKE Operator](#) section. All platforms support the percent sign (%) for a wildcard symbol. For example, to find any phone number starting with the 415 area code:

```
WHERE phone LIKE '415%'
```

EXISTS

Used only with subqueries, *EXISTS* tests to see whether the subquery data exists. It is typically much faster than *WHERE IN* subqueries. For example, the following query finds all authors who are also employees:

```
SELECT au_lname FROM authors WHERE EXISTS (SELECT last_name FROM employees)
```

BETWEEN

Performs a range check to see if a value is in between two values (inclusive of those two values). For example:

```
WHERE ytd_sales BETWEEN 4000 AND 9000.
```

IN

Performs a test to see if an expression matches any one value out of a list of values. The list may be literal, as in *WHERE state IN ('or', 'il', 'tn', 'ak')*. The list of values may also be derived using a subquery:

```
WHERE state IN (SELECT state_abbr FROM territories).
```

SOME | ANY

Functions the same as the *EXISTS* operation, though with a slightly different syntax. For example, the following query finds all authors who are also employees:

```
SELECT au_lname FROM authors WHERE au_lname = SOME(SELECT last_name FROM employees)
```

ALL

Performs a check to see if all records in the subquery match the evaluation criteria. It returns TRUE when the subquery returns zero rows. For example:

```
WHERE city = ALL (SELECT city FROM employees WHERE emp_id = 54123)
```

The GROUP BY clause

The *GROUP BY* clause (and the *HAVING* clause) is needed only in queries that utilize *aggregate functions*.

The *GROUP BY* clause is used to report an aggregated value for one or more rows returned by a *SELECT* statement based on one or more non-aggregated columns called *grouping columns*. For example, here is a query that counts up how many people we hired per year from the year 1999 to 2004:

```
SELECT hire_year, COUNT(emp_id) AS 'nbr_emps'  
  
FROM employee  
  
WHERE status = 'ACTIVE'  
  
    AND hire_year BETWEEN 1999 AND 2004  
  
GROUP BY hire_year;
```

The results are:

```
hire_year  nbr_emps  
-----  
1999      27  
2000      17  
2001      13  
2002      19  
2003      20  
2004      32
```

Queries using aggregate functions provide many types of summary information. The most common aggregate functions include:

AVG

Returns the average of all non-NULL values in the specified column(s).

AVG DISTINCT

Returns the average of all unique, non-NULL values in the specified column(s).

COUNT

Counts the occurrences of all non-NULL values in the specified column(s).

COUNT DISTINCT

Counts the occurrences of all unique, non-NULL values in the specified column(s).

COUNT()*

Counts every record in the table.

MAX

Returns the highest non-NULL value in the specified column(s).

MIN

Returns the lowest non-NULL value in the specified column(s).

SUM

Totals all non-NULL values in the specified column(s).

SUM DISTINCT

Totals all unique, non-NULL values in the specified column(s).

Some queries that use aggregates return a sole value. Single value aggregates are known as a *scalar aggregates*. Scalar aggregates do not need a *GROUP BY* clause. For example:

--Query

```
SELECT AVG(price)
```

```
FROM titles
```

--Results

```
14.77
```

Queries that return both regular column values and aggregate function values are commonly called *vector aggregates*. Vector aggregates use the *GROUP BY* clause and return one or many rows. There are a few rules to follow when using *GROUP BY*:

- Place *GROUP BY* in the proper clause order—after the *WHERE* clause and before the *ORDER BY* clause.
- Include all non-aggregate columns in the *GROUP BY* clause.
- Do not use a column alias in the *GROUP BY* clause, though table aliases are acceptable.

For example, let's suppose you need to know the total purchase amount of several purchases, with an **Order_Details** table that looks like this:

OrderID	ProductID	UnitPrice	Quantity
10248	11	14.0000	12
10248	42	9.8000	10
10248	72	34.8000	5
10249	14	18.6000	9
10249	51	42.4000	40
10250	41	7.7000	10
10250	51	42.4000	35
10250	65	16.8000	15
...			

The following example will give you the results:

```
SELECT OrderID, SUM(UnitPrice * Quantity) AS 'Order Amt'  
FROM order_details  
WHERE orderid IN (10248, 10249, 10250)  
GROUP BY orderid
```

The results are:

OrderID	Order Amt
10248	440.0000
10249	1863.4000
10250	1813.0000

We could further refine the aggregations by using more than one grouping columns. Considering the following query retrieving the average price of our products, grouped first by name and then by size:

```
SELECT name, size, AVG(unit_price) AS 'avg'  
FROM product  
GROUP BY name, size
```

The results are:

Name	Size	avg
Flux Capacitor	small	900
P32 Space Modulator	small	1400
Transmogrifier	medium	1400
Acme Rocket	large	600
Land Speeder	large	6500

In addition, the *GROUP BY* clause supports a number of very important subclauses:

```
GROUP BY [ {ROLLUP | CUBE} ] ( [ grouping_column [,...] ] ) [ , grouping set list]
```

Groups the aggregate values of the result set by one or more grouping columns. (Without *ROLLUP* or *CUBE*, the *GROUP BY* (*grouping_column* [,...]) clause is the most simplest and common form of the *GROUP BY* clause.)

ROLLUP

Produces subtotals for each set of grouping columns as a hierarchical result set. It adds subtotal and grand total rows into the result sets in a hierarchical fashion. *ROLLUP* operations return one row per grouping column where NULL appears in the grouping column to show the subtotaled or totaled aggregate value (illustrated in a moment).

CUBE

Produces subtotals and cross-tabulated totals for all grouping columns. In a sense, the *CUBE* clause enables you to quickly return multidimensional result sets from standard relational tables without a lot of programmatic work. *CUBE* is especially useful when working with large amounts of data. Like *ROLLUP*, *CUBE* provides subtotals of the grouping columns but it also includes subtotal rows for all possible combinations of the grouping columns specified in the query.

```
GROUP BY GROUPING SETS [ {ROLLUP | CUBE} ] ( [ grouping_column [,...] ] ) [ , grouping set list]
```

Enables aggregated groups on several different sets of grouping columns within the same query. This is especially useful when you want to return only a portion of an aggregated result set. The *GROUPING SETS* clause also lets you select which grouping columns to compare, whereas *CUBE* returns all of the grouping columns and *ROLLUP* returns a hierarchical subset of the grouping columns. As the syntax shows, the ANSI standard also allows *GROUPING SETS* to be paired with *ROLLUP* or *CUBE*.

Consider the following clauses and the result sets they return, shown in [Table 3-3](#).

Table 3-3. GROUP BY syntax variations

<i>GROUP BY</i> syntax	Returns the following sets
<i>GROUP BY</i> (col_A, col_B, col_C)	(col_A, col_B, col_C)
<i>GROUP BY ROLLUP</i> (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) ()
<i>GROUP BY CUBE</i> (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) (col_B, col_C) (col_B) (col_C) ()
<i>GROUP BY GROUPING SETS</i> ((col_A, col_B), (col_A, col_C), (col_C))	Subquery: SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')

Each type of *GROUP BY* clause returns a different set of aggregated values and, in the case of *ROLLUP* and *CUBE*, totals and subtotals.

The concepts of *ROLLUP*, *CUBE*, and *GROUPING SETS* are much more intuitive when explained by example. In the following example, we query for data summarizing the number of **sales_orders** by **order_year** and by **order_quarter**:

```
SELECT order_year AS year, order_quarter AS quarter,
       COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY ROLLUP (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

The results are:

```
year quarter orders
```

```
-----
```

```
NULL NULL 648 -- the grand total
2003 NULL 380 -- the total for year 2003
2003 1 87
2003 2 77
2003 3 91
2003 4 125
2004 NULL 268 -- the total for year 2004
2004 1 139
2004 2 119
2004 3 10
```

Adding grouping columns to the query provides more details (and more subtotaling) in the result set. We'll modify the previous example by adding a **region** to the query (but since the number of rows increases, we'll only look at the first and second quarter):

```
SELECT order_year AS year, order_quarter AS quarter, region,
       COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
      AND order_quarter IN (1,2)
      AND region IN ('USA', 'CANADA')
GROUP BY ROLLUP (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

The results are:

```
year quarter region orders
```

```
-----
```

```
NULL NULL NULL 183 -- the grand total
2003 NULL NULL 68 -- the subtotal for year 2003
2003 1 NULL 36 -- the subtotal for all regions in q1 of 2003
2003 1 CANADA 3
2003 1 USA 33
2003 2 NULL 32 -- the subtotal for all regions in q2 of 2004
2003 2 CANADA 3
2003 2 USA 29
2004 NULL NULL 115 -- the subtotal for year 2004
2004 1 NULL 57 -- the subtotal for all regions in q1 of 2004
```



```
2004 1    CANADA  11
2004 1    USA     46
2004 2    NULL   58    -- the subtotal for all regions in q2 of 2004
2004 2    CANADA   4
2004 2    USA    54
```

The *GROUP BY CUBE* clause is useful for performing multidimensional analysis on aggregated data. Like *GROUP BY ROLLUP*, it returns subtotals combining all of the grouping columns named in the query. (As you will see, it also has the potential to increase the number of rows returned in the result set.)

In the following example, we query for data summarizing the number of **sales_orders** by **order_year** and by **order_quarter**:

```
SELECT order_year AS year, order_quarter AS quarter,
       COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY CUBE (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

The results are:

```
year quarter orders
```

```
----
```

```
NULL NULL   648    -- the grand total
NULL 1     226    -- the subtotal for q1 of both years
NULL 2     196    -- the subtotal for q2 of both years
NULL 3     101    -- the subtotal for q3 of both years
NULL 4     125    -- the subtotal for q4 of both years
2003 NULL   380    -- the total for year 2003
2003 1      87
2003 2      77
2003 3      91
2003 4     125
2004 NULL   268    -- the total for year 2004
2004 1     139
2004 2     119
2004 3      10
```

The *GROUP BY GROUPING SETS* clause lets you aggregate on more than one group in a single query. For each group set, the query returns subtotals with the grouping column marked as NULL. While the *CUBE* and *ROLLUP* clauses place predefined subtotals into the result set, the *GROUPING SETS* clause allows you to control what subtotals to add to the query. The *GROUPING SETS* clause does not return a grand total.

Using a similar example query to the ones shown with *ROLLUP* and *CUBE*, this time we chose to subtotal by **year** and **quarter** and separately by **year**:

```
SELECT order_year AS year, order_quarter AS quarter, COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY GROUPING SETS ( (order_year, order_quarter), (order_year) )
ORDER BY order_year, order_quarter;
```

The results are:

```
year quarter orders
-----
2003 NULL 380 -- the total for year 2003
2003 1 87
2003 2 77
2003 3 91
2003 4 125
2004 NULL 268 -- the total for year 2004
2004 1 139
2004 2 119
2004 3 10
```

Another way to think of *GROUPING SETS* is to consider them to be like a *UNION ALL* of more than one *GROUP BY* query that references different parts of the same data. You can tell the database to add subtotals to a *GROUPING SET* by simply adding in the *ROLLUP* or *CUBE* clause according to how you would like subtotaling to occur.

GROUPING SETS can also be concatenated to concisely generate large combinations of groupings. *Concatenated GROUPING SETS* yield the cross-product of groupings from each of the sets within a *GROUPING SET* list. Concatenated *GROUPING SETS* are compatible with *CUBE* and *ROLLUP*. Concatenated *GROUPING SETS*, since they perform a cross-product of all *GROUPING SETS*, will generate a very large number of final groupings from even a small number of concatenated groupings.

Table 3-4. GROUP BY syntax variations

<i>GROUP BY</i> syntax	Returns the following sets
<i>GROUP BY</i> (col_A, col_B, col_C)	(col_A, col_B, col_C)
...	...
<i>GROUP BY GROUPING SETS</i> (col_A, col_B) (col_Y, col_Z)	(col_A, col_Y) (col_A, col_Z) (col_B, col_Y) (col_B, col_Z)

In the example in [Table 3-4](#), the concatenated *GROUPING SETS* yield a large number of final groupings. You can imagine how large the result set would be if the concatenated *GROUPING SETS* contained a large number of groupings! However, the information returned can be very valuable and hard to reproduce.

The HAVING clause

The *HAVING* clause adds search conditions on the result of the *GROUP BY* clause. *HAVING* works very much like the *WHERE* clause, but applies to the *GROUP BY* clause. The *HAVING* clause supports all the same search conditions as the *WHERE* clause shown earlier. For example, using the same query as before, we now want to find only those jobs that have more than three people:

--Query

```
SELECT  j.job_desc "Job Description",
        COUNT(e.job_id) "Nbr in Job"
FROM    employee e
JOIN    jobs j ON e.job_id = j.job_id
GROUP BY j.job_desc
HAVING  COUNT(e.job_id) > 3
```

--Results

Job Description	Nbr in Job
Acquisitions Manager	4
Managing Editor	4
Marketing Manager	4
Operations Manager	4
Productions Manager	4
Public Relations Manager	4
Publisher	7

Note that the ANSI standard does not *require* that an explicit *GROUP BY* clause appear with a *HAVING* clause. For example, the following query against the **employee** table is valid because it has an implied *GROUP BY* clause:

```
SELECT COUNT(dept_nbr)
FROM employee
HAVING COUNT(dept_nbr) > 30;
```

Although the earlier code example is valid, this application for the *HAVING* clause is rather rare.

The ORDER BY clause

A result set can be sorted through the *ORDER BY* clause, in accordance with the database's sort order. Each column of the result set may be sorted in either ascending (*ASC*) or descending (*DESC*) order. (Ascending order is the default.) If no *ORDER BY* clause is specified, most implementations return the data according to the physical order of data within the table or according to the order of an index utilized by the query. However, when no *ORDER BY* clause is specified, there is no guarantee on the order of the result set. Following is an example of a *SELECT* statement with an *ORDER BY* clause on SQL Server:

```
SELECT  e.emp_id "Emp ID",
        e.fname "First",
        e.lname "Last",
        j.job_desc "Job Desc"
FROM    employee e,
        jobs j
```

```
WHERE e.job_id = j.job_id
      AND j.job_desc = 'Acquisitions Manager'
ORDER BY e.ename DESC,
         e.lname ASC
```

The results are:

Emp ID	First	Last	Job Desc
MIR38834F	Margaret	Rancé	Acquisitions Manager
MAS70474F	Margaret	Smith	Acquisitions Manager
KJJ92907F	Karla	Jablonski	Acquisitions Manager
GHT50241M	Gary	Thomas	Acquisitions Manager

After the result set is pared down to meet the search conditions, the result set is sorted by the authors' first names in descending order. Where the authors' first names are equal, the results are sorted in ascending order by last name.



You may write an *ORDER BY* clause using columns in the table that do not appear in the select list. For example, you might query all **emp_ids** from the employee table, yet *ORDER BY* the employee's first and last name.

Programming Tips and Gotchas

Once an alias has been assigned to a table or view in the *FROM* clause, use it exclusively for all other references to that table or view within the query (in the *WHERE* clause, for example). Do *not* mix references to the full table name and the alias within a single query. You should avoid mixed references for a couple of reasons. First, it is simply inconsistent and makes code maintenance more difficult. Second, some database platforms return errors on *SELECT* statements containing mixed references. (Refer to the [SUBQUERY Substatement](#) section for special instructions on aliasing within a subquery.)

MySQL, PostgreSQL, and SQL Server support certain types of queries that do not need a *FROM* clause. Use these types of queries with caution since the ANSI standard requires a *FROM* clause. Queries without a *FROM* clause must be manually migrated to either the ANSI standard form or to a form that also works on the target database. Certain platforms do not support the ANSI style *JOIN* clause. Refer to the entry for each clause to fully investigate the varying degrees of support among the different database vendors for all the options of the *SELECT* command.

DB2

DB2 supports all of the typical clauses of the ANSI standard *SELECT* statement, such as *WHERE*, *FROM*, *GROUP BY*, *HAVING*, and *ORDER BY*. DB2 supports all of the subclauses of the *GROUP BY* clause, including concatenated *GROUPING SETS*. Plus, DB2 supplements the most common clauses with its own *SELECT INTO* clause and *FETCH FIRST* clause:

```
SELECT { [ALL | DISTINCT] } select_item [AS alias] [...]  
  
[INTO host_variable [...] ]  
  
FROM [ONLY | OUTER] {table_name | view_name | TABLE ( {function_name |  
    subquery} ) [AS alias] [...] }  
  
[ [join type] JOIN table2 ]  
  
[WHERE search_condition] [ {AND | OR | NOT} search_condition [...] ]  
  
[GROUP BY group_by_expression [HAVING search_condition] ]
```

[ORDER BY *order_expression* [ASC | DESC]]

[FETCH FIRST {*n*} {ROW | ROWS} ONLY]

where:

INTO *host_variable* [...]

Assigns the values returned by the query to the host variable(s). An error will occur if the *SELECT* statement returns more than one row in the result set. The values of each *select_item* are assigned in ordinal position to the declared *host_variables*. A warning is given if there are fewer host variables than values returned by the *SELECT* statement.

TABLE ({*function_name* | *subquery*})

Specifies that the query will return a function result set or the results of a fully declared subquery. When making a table function call, the column names of the table function specified in the *RETURNS* clause of the *CREATE FUNCTION* statement are analogous the columns of the table. Note that the function call, along with its parameters and the subquery, must be enclosed in parentheses. For example, the following query takes data from an international retail web site and returns the conversions between foreign currencies and the U.S. dollar:

SELECT column1

FROM TABLE (curr_conv(CURRENCY, CONVERSION_RATE)) AS data;

GROUP BY *group_by_expression*

DB2 fully supports the ANSI standard *GROUP BY* clause, including *ROLLUP*, *CUBE*, and *GROUPING SETS*.

FETCH FIRST {*n*} {ROW | ROWS} ONLY

Sets a maximum number of rows that will be retrieved by the query. This clause is used to tell the database that you do not want more than *n* rows, where *n* is any positive integer of 1 or greater, regardless of how many might be retrieved by the database. This can improve performance, since potentially large result sets may be trimmed to a more manageable size.

DB2 supports four possible join types: *INNER*, *LEFT OUTER*, *RIGHT OUTER*, and *FULL OUTER*:

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* with join conditions in the *WHERE* clause is semantically equivalent to an inner join.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all the records in a table even when there are no counterparts in the joined table. For example:

SELECT j.job_id, e.lname

FROM jobs j

LEFT OUTER JOIN employee e ON j.job_id = e.job_id

ORDER BY j.job_id

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. A right join is the same as a left join, except that the optional table is on the left.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

DB2 places some restrictions on using an *ORDER BY* clause or a *FETCH FIRST* clause in any *SELECT* statement used in a subquery. First, a *SELECT* statement containing an *ORDER BY* clause or a *FETCH FIRST* clause must always be enclosed in parentheses, even when used in a set operation. Second, a *SELECT* statement containing an *ORDER BY* clause or a *FETCH FIRST* clause will return an error if it is the primary query of a view or a materialized query table, since views and materialized query tables cannot be inherently ordered.

The *FETCH FIRST* clause can be used to do some interesting things beyond merely limiting the total number of records returned by a result set. For example, the following query does not limit the total number of records returned by the query; instead, it limits only the number of records used in the subquery. In this example, we will return all the titles of the top 10 employees, based on their salaries:

```
SELECT jobs.job_desc
FROM jobs
WHERE jobs.job_desc IN
    (SELECT employee.job_desc
     FROM employee
     ORDER BY salary DESC
     FETCH FIRST 10 ROWS ONLY);
```

In the example above, only the first 10 rows of the employee table will be used, but the parent query might possibly return any number of rows. Also remember that the ordering in the subquery has no impact on the ordering of the final result set.

MySQL

MySQL's implementation of *SELECT* includes partial *JOIN* support, the *INTO* clause, the *LIMIT* clause, and the *PROCEDURE* clause. MySQL does *not* support subqueries prior to Version 4.0.

```
SELECT [DISTINCT |
DISTINCTROW | ALL]
    [STRAIGHT_JOIN] [ {SQL_SMALL_RESULT | SQL_BIG_RESULT} ] [SQL_BUFFER_RESULT]
    [HIGH_PRIORITY]
select_item AS alias [,...]
[INTO {OUTFILE | DUMPFILE} 'file_name' options]
[FROM table_name AS alias [,...]
    [ { USE INDEX (index1 [,...]) | IGNORE INDEX (index1 [,...]) } ]
[join type][JOIN table2] [ON join_condition]
[WHERE search_condition]
[GROUP BY {unsigned_integer | column_name | formula}
    [ASC | DESC] [,...]]
[HAVING search_condition]
[ORDER BY {unsigned_integer | column_name | formula}]
```

[ASC | DESC] [,...]]

[LIMIT [[*offset_position*,] *number_of_rows*]]

[PROCEDURE *procedure_name* (*param* [,...])]

[{FOR UPDATE | LOCK IN SHARE MODE}];

where:

STRAIGHT_JOIN

Forces the optimizer to join tables in the exact order they appear in the *FROM* clause.

SQL_SMALL_RESULT | *SQL_BIG_RESULT*

Tells the optimizer to expect a small or large result set on a *GROUP BY* clause or a *DISTINCT* clause, respectively. MySQL builds a temporary table when a query has a *DISTINCT* or *GROUP BY* clause, and these optional clauses tell MySQL to build a fast temporary table in memory (for *SQL_SMALL_RESULT*) or a slower, disk-based temporary table (for *SQL_BIG_RESULT*) to process the worktable.

SQL_BUFFER_RESULT

Forces the result set into a temporary table so that MySQL can free table locks earlier and speed the result set to the client.

HIGH_PRIORITY

Gives the query a higher priority than statements that modify data within the table. This should be used only for special, high-speed queries.

select_item

Retrieves the expressions or columns listed. Columns may be in the format [*database_name*.] [*table_name*.]*column_name*. If the database or table names are left out, then MySQL assumes the current database and table.

FROM ... USE INDEX | *IGNORE INDEX*

Indicates the table from which rows will be retrieved. The table may be described as [*database_name*.] [*table_name*]. MySQL will treat the query as a join if more than one table appears in the *FROM* clause. The optional *FORCE INDEX* and *USE INDEX* (*index1*, *index2*, ...), and *IGNORE INDEX* (*index1*, *index2*, ...) clauses enable you to tell MySQL to use or ignore specific indexes on a table, respectively.

INTO {OUTFILE | DUMPFILE}'*file_name*'

Writes the result set of the query to a file named '*file_name*' on the host filesystem with the *OUTPUT* option. The *file_name* must not already exist. The *DUMPFIL*E option writes a single continuous line of data without column terminations, line terminations, or escape characters. This option is used mostly for *BLOB* files. Specific rules for using this clause are detailed below.

LIMIT [[*offset_position*,] *number_of_rows*]

Constrains the number of rows returned by the query, starting at the *offset_position* and returning *number_of_rows*. If only one integer is supplied, then the specified number of records is returned, and a default offset (or starting position) of zero is assumed.

PROCEDURE *procedure_name* (*param* [,...])

Names a procedure that processes the data in the result set. The procedure is an external procedure, usually C++, not an internal database stored procedure.

{FOR UPDATE | LOCK IN SHARE MODE}

Issues either a write lock on the rows returned by the query (*FOR UPDATE*) for its exclusive use (provided the table is of InnoDB or BDB type) or issues read locks on the rows returned by the query (*LOCK IN SHARE*) such that other users may see the rows, but may not modify them.

Keep a couple of rules in mind when using the *INTO* clause. First, the output file cannot already exist since overwrite functionality is not supported. Second, any file created by the query will be readable by everyone that can connect to the server. (When using *SELECT...INTO OUTFILE*, you can then turn around and use the MySQL command *LOAD DATA INFILE* to quickly load the data.)

You can use the following options to better control the content of the output file when using *SELECT...INTO OUTFILE*:

- *ESCAPED BY*
- *FIELDS TERMINATED BY*
- *LINES TERMINATED BY*
- *OPTIONALLY ENCLOSED BY*

The following example illustrates the use of these optional commands via a MySQL query that returns a result set in a comma-delimited output file:

```
SELECT job_id, emp_id, lname+fname
INTO OUTFILE "/tmp/employees.text"
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY "\n"
FROM employee;
```

MySQL also allows *SELECT* statements without a *FROM* clause when performing simple arithmetic. For example, the following queries are valid on MySQL:

```
SELECT 2 + 2;
SELECT 565 - 200;
SELECT (365 * 2) * 52;
```

MySQL is very fluid in the way it supports joins. You can use several different syntaxes to perform a join. For example, you can explicitly declare a join in a query using the *JOIN* clause, but then show the join condition in the *WHERE* clause. The other platforms force you to pick one method or the other and do not allow you to mix them in a single query. However, we think it's bad practice to mix methods, so our examples use ANSI *JOIN* syntax.

MySQL supports the following types of *JOIN* syntax:

```
FROM table1, table2
FROM table1 {
  [CROSS] JOIN table2 |
  STRAIGHT_JOIN table2 |
  INNER JOIN table2 [ {ON join_condition |
  USING (column_list)} ] |
  LEFT [OUTER] JOIN table2 {ON join_condition |
  USING (column_list)} |
  NATURAL [LEFT [OUTER]] JOIN table2 |
  RIGHT [OUTER] JOIN table2 [ {ON join_condition |
```


USING (column_list)]]

NATURAL [RIGHT [OUTER]] JOIN table2 }

where:

[CROSS] JOIN table2

Retrieves all records from both *table1* and *table2*. Refer to the [JOIN Subclause](#) section on "Joins" for more information.

STRAIGHT_JOIN table2

Retrieves all records of both *table1* and *table2* as any other *JOIN* statement would, with some special behavior. Normally, MySQL evaluates the table order of a join operation based on choices made by the query optimizer. However, the optimizer can occasionally choose an ineffective table order. The *STRAIGHT_JOIN* keyword forces the optimizer to evaluate the tables in the order listed in the query.

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* is semantically equivalent to an inner join. When using *FROM table1, table2*, be sure include the join condition(s) in the *WHERE* clause.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause that names all columns that exist in both tables. (An example of the *USING* clause appears later.)

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all records in a table even when there are no counterparts in the joined table.

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. A *RIGHT JOIN* is nothing more than a *LEFT JOIN* written from the opposite direction. As a best practice, write all outer join queries as either *LEFT* or *RIGHT OUTER JOIN* queries, but do not use both types, since they require a shift in thinking to understand at first glance.

MySQL offers an interesting alternative to the ANSI SQL standard for querying tables — the *HANDLER* statement. The *HANDLER* statement works a lot like *SELECT*, except that *HANDLER* provides very rapid data reads that circumvent the SQL query engine in MySQL. However, since the *HANDLER* statement is not a SQL statement, refer to the MySQL documentation for more information.

Oracle

Oracle allows a very large number of extensions to the ANSI *SELECT* statement. For example, since both nested tables and partitioned tables are allowed in Oracle (see *CREATE TABLE*), the *SELECT* statement allows queries to those types of structures.

[WITH query_name AS (subquery) [...]]

SELECT { {[ALL | DISTINCT]} | [UNIQUE]} [optimizer_hints]

select_item [AS alias] [...]

[INTO {variable [...] | record}]

FROM {[ONLY] {[schema.][table_name| view_name |

materialized_view_name]}[@database_link]

```
[AS [OF] {SCN | TIMESTAMP} expression] |
subquery [WITH
  {READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]}] |
[[VERSIONS BETWEEN {SCN | TIMESTAMP} {exp | MINVALUE AND
  {exp | MAXVALUE}] AS OF {SCN | TIMESTAMP} expression] |
TABLE (nested_table_column) [(+)]
  {[PARTITION (partition_name) |
  SUBPARTITION (subpartition_name)]}
[SAMPLE [BLOCK] [sample_percentage] [SEED (seed_value)]] [AS alias]
[,...]
[ [join_type] JOIN join_condition [PARTITION BY expression [...]] ]
[WHERE search_condition]
[ {AND | OR} search_condition [...]]
[[START WITH value] CONNECT BY [PRIOR] condition] ]
[GROUP BY group_by_expression [HAVING search_condition] ]
[MODEL model_clause]
[ORDER [SIBLINGS] BY order_expression {[ASC | DESC]}
  {NULLS FIRST | NULLS LAST}] ]
[FOR UPDATE [OF [schema.][table.]column] [...]]
  {[NOWAIT | WAIT (integer)]} ]
```

Unless otherwise noted, the clauses shown above follow the ANSI standard. Similarly, elements of the clauses are identical to the ANSI standard unless otherwise noted. For example, Oracle's *GROUP BY* clause is nearly identical to the ANSI standard, including its component elements like *ROLLUP*, *CUBE*, *GROUPING SETS*, concatenated *GROUPING SETS*, and the *HAVING* clause.

WITH *query_name* AS (*subquery*) [...]

References a named query. Oracle allows you to assign names to subqueries. You can then reference the subquery in multiple places by specifying only its name. (Oracle optimizes this process by treating the subquery as a nested table subquery or a temporary table.)

ALL | DISTINCT | UNIQUE

ALL and DISTINCT are identical to the ANSI *ALL* and *DISTINCT* clauses (see the ANSI SQL syntax earlier). *UNIQUE* is a synonym for *DISTINCT*. *DISTINCT* cannot be used on [LOB](#) columns.

optimizer_hint

Overrides the default behavior of the query optimizer with user-specified behaviors. For example, hints can force Oracle to use an index that it might not otherwise use or to avoid an index that it might otherwise use. Refer to the vendor documentation for more information about optimizer hints.

select_item

May be an expression or a column from a named query, table, view, or materialized view in the format [*schema*.]*[table_name.] column_name*. If you omit the schema, Oracle assumes the context of the current schema. Oracle also allows for *named queries* (explained under the *WITH* keyword above) that may be referenced much like a nested table subquery (refer to the [SUBQUERY Substatement](#) section). Oracle refers to named queries as *subquery factoring*. In addition to named queries, Oracle also supports subqueries and the asterisk (*),

shorthand for all columns, in the *select_item* list.

INTO {*variable* [, ...] | *record*}

Retrieves the result set values into PL/SQL variables or into a PL/SQL record.

FROM [ONLY]

Identifies the table, view, materialized view, partition, or subquery from which the result set is retrieved. The *ONLY* keyword is optional and only applies to views belonging to a hierarchy. Use *ONLY* when you want to retrieve records from a named view only, and not from any of its subviews.

AS [OF] {SCN | TIMESTAMP} *expression*

Implements SQL-driven flashback whereby system change numbers (or timestamps) are applied to each object in the *select_list*. Records retrieved by the query are only those that existed at the specific system change number (SCN) or time. (This feature can also be implemented at a session-level using the *DBMS_FLASHBACK* built-in package.) *SCN expression* must equal a number while *TIMESTAMP expression* must equal a timestamp value. Flashback queries cannot be used on linked servers.

subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT *constraint_name*]}]

Mentioned separately because Oracle allows you extra ways to control a subquery. *WITH READ ONLY* indicates that the target of the subquery cannot be updated. *WITH CHECK OPTION* indicates any update to the target of the subquery must produce rows that would be included in the subquery. *WITH CONSTRAINT* creates a *CHECK OPTION* constraint of *constraint_name* on the table. Note that *WITH CHECK OPTION* and *WITH CONSTRAINT* are usually used in *INSERT...SELECT* statements.

VERSIONS BETWEEN {SCN | TIMESTAMP} {*exp* | MINVALUE AND {*exp* | MAXVALUE}] AS OF {SCN | TIMESTAMP} *expression*]

Specifies a special kind of query using a *flashback_query_clause* to retrieve the history of changes made to data from a table, view, or materialized view. The *VERSIONS_XID* pseudocolumn shows the identifier corresponding to the transaction that made the change. A *flashback_query_clause* requires that you specify an *SCN* (system change number) or a *TIMESTAMP* value for each object in the *select_item* list. (Implement SQL-driven session-level flashback using the Oracle *DBMS_FLASHBACK* package.)

The optional subclause *VERSIONS BETWEEN* is used to retrieve multiple versions of the data specified, using either an upper and lower boundary of an *SCN* (a number) or *TIMESTAMP* (a timestamp value), or by using the *MINVALUE* and *MAXVALUE* keywords. Without this clause, only one past version of the data is returned. (Oracle also provides several version query pseudocolumns for additional versioning information.)

The *AS OF* clause, discussed earlier in this list, determines the SCN or moment in time from which the database issues the query when used with the *VERSIONS* clause.

You cannot use flashback queries with the *VERSIONS* clause against temporary tables, external tables, tables in a cluster, or views.

TABLE

Required when querying a hierarchically declared nested table.

PARTITION

Restricts a query to the specified partition of the table. In essence, rows are retrieved only from the named partition, not the entire table.

SUBPARTITION

Restricts a query to the specified subpartition of the table. I/O is reduced because the rows are retrieved only from the named subpartition instead of the entire table.

SAMPLE [BLOCK] [*sampling_percentage*] [SEED (*seed_value*)]

Tells Oracle to select records from a random sampling of rows, as either a percentage of rows or blocks, within the result set rather than from the entire table. *BLOCK* tells Oracle to use block sampling rather than row sampling. The sampling percentage, telling Oracle the total block or row percentage to be included in the sample, may be anywhere between .000001 and 99. The optional *SEED* clause is used to provide limited repeatability. By specifying a seed value, Oracle will attempt to return the same sample from one execution of the query to the next. The seed value can be between 0 and 4294967295. When *SEED* is omitted, the resulting sample will change from one execution of the query to the next. Sampling may be used only on single-table queries.

JOIN

Merges the result sets of two or more tables in a single query. Described in detail below.

PARTITION BY *expression* [...]

Defines a special kind of query called a *partitioned outer join* that extends the conventional outer join syntax by applying a right or left outer join to a partition of one or more rows specified by the *expression*. This is especially useful for querying sparse data along a particular dimension of data thereby returning rows that otherwise would be omitted from the result set. For an example, see the "Oracle" subsection under [JOIN Subclause](#). The *PARTITION BY* clause can be used on either side of an outer join, resulting in a *UNION* of the outer joins of each of the partitions in the partitioned result set and the table on the other side of the join. (When this clause is omitted, Oracle treats the entire result set as a single partition.) *PARTITION BY* is not allowed with *FULL OUTER JOIN*.

WHERE ... [[START WITH *value*] CONNECT BY [PRIOR] *condition*]

Filters records returned in the result set. Oracle allows the use of hierarchical information within tables, whose filtering can be controlled with the *START WITH* clause. *START WITH* identifies the rows that will serve as the parent rows in the result set. *CONNECT BY* identifies the relationship *condition* between the parent rows and their child rows. The *PRIOR* keyword is used to identify the parent rows instead of the child rows.

Hierarchical queries use the *LEVEL* pseudocolumn to identify the root node (1), the child nodes (2), the grandchild nodes (3), and so forth. Other pseudocolumns available in hierarchical queries are *CONNECT_BY_ISCYCLE* and *CONNECT_BY_ISLEAF*. Hierarchical queries are mutually exclusive of the *ORDER BY* and *GROUP BY* clauses. Do not use those clauses in a query containing *START WITH* or *CONNECT BY*. You can order records from siblings of the same parent table by using the *ORDER SIBLINGS BY* clause.

ORDER [SIBLINGS] BY *order_expression* {NULLS FIRST | NULLS LAST}}

Sorts the result set of the query in order of *order_expression*. The *order_expression* may be the column names, their aliases, or integers indicating their ordinal position. The *ORDER SIBLINGS BY* clause is used when querying hierarchical tables with the *CONNECT BY* clause. *ORDER SIBLINGS BY* tells Oracle to preserve any ordering specified in the *CONNECT BY* clause and applies the ordering only to siblings (i.e., rows of equal level in the hierarchy). *NULLS FIRST* and *NULLS LAST* specify that the records containing *NULLS* should appear either first or last, respectively.

FOR UPDATE [OF [*schema*.]...]

Locks the rows of the result set so that other users cannot lock or update them until you're finished with your transaction. *FOR UPDATE* cannot be used in a subquery, in queries using *DISTINCT* or *GROUP BY*, or in queries with set operators or aggregate functions. Child rows in a hierarchical table are not locked when issued against the parent rows. The *OF* keyword is used to lock only the selected table or view. Otherwise, Oracle locks all the tables or views referenced in the *FROM* clause. When using *OF*, the columns are not significant, though real column names and not aliases must be used. The *NOWAIT* and *WAIT* keywords tell Oracle to either return control immediately if a lock already exists or to wait *integer* seconds before returning control to you, respectively. If neither *NOWAIT* nor *WAIT* is specified, Oracle waits until the rows become available.

Unlike some other database platforms, Oracle does not allow a *SELECT* statement without a *FROM* clause. The following query, for example, is invalid:

```
SELECT 2 + 2;
```

In order to work around this, Oracle has provided a special-purpose table called **dual**. Any time you want to write a query that does not retrieve data from a user-created table, such as to perform a calculation, use *FROM dual*. All of the following queries are valid:

```
SELECT 2 + 2  
  
FROM dual;  
  
SELECT (((52-4) * 5) * 8)  
  
FROM dual;
```

Oracle's implementation of *SELECT* is quite straightforward if you want to retrieve data from a table. For example, Oracle allows the use of *named queries*. A named query is, in a sense, an alias to an entire query that can save you time when writing a complex multi-subquery *SELECT* statement. For example:

```
WITH pub_costs AS  
  
    (SELECT pub_id, SUM(job_lvl) dept_total  
  
     FROM employees e  
  
     GROUP BY pub_id),  
avg_costs AS  
  
    (SELECT SUM(dept_total)/COUNT(*) avg  
  
     FROM employeee)  
SELECT * FROM pub_costs  
  
WHERE dept_total > (SELECT avg FROM avg_cost)  
  
ORDER BY department_name;
```

In this example, we create two named subqueries — **pub_costs** and **avg_costs** — which are later referenced in the main query. The named queries are effectively the same as subqueries; however, subqueries must be written out in their entirety each time they're used while named queries need not.

Oracle allows you to select rows from a single partition of a partitioned table (using the *PARTITION* clause) or to retrieve only a statistical sampling of the rows, as a percentage of rows or blocks, of a results set using *SAMPLE*. For example:

```
SELECT *  
  
FROM sales PARTITION (sales_2004_q3) sales  
  
WHERE sales.qty > 1000;  
  
SELECT *  
  
FROM sales SAMPLE (12);
```

Flashback queries are a feature of Oracle that enable retrieval of point-in-time result sets. For example, you could find out what everyone's salary was yesterday before a big change was applied to the database:

```
SELECT job_lvl, lname, fname  
  
FROM employee  
  
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY);
```

Another interesting Oracle extension of the standard query format is the *hierarchic query*. Hierarchic queries return the results of queries against hierarchically designed tables in the order you define. For example, the following query returns the names of the employees, their position in the hierarchy (represented by the position in the **org_char** column), the employee ID, the manager ID, and their job ID:

```
SELECT LPAD(' ',2*(LEVEL-1)) || lname AS org_chart,  
  
       emp_id, mgr_id, job_id
```

```
FROM employee

START WITH job_id = 'Chief Executive Officer'

CONNECT BY PRIOR emp_id = mgr_id;

ORG_CHART      EMPLOYEE_ID  MANAGER_ID  JOB_ID
-----
Cramer         101          100 Chief Executive Officer
Devon          108          101 Business Operations Mgr
  Thomas       109          108 Acquisitions Manager
  Koskitalo    110          108 Productions Manager
  Tonini       111          108 Operations Manager
Whalen         200          101 Admin Assistant
Chang          203          101 Chief Financial Officer
  Gietz       206          203 Comptroller
Buchanan       102          101 VP Sales
  Callahan    103          102 Marketing Manager
```

In the previous query, the *CONNECT BY* clause defines the hierarchical relationship of the **emp_id** value as the parent row equal to the **mgr_id** value in the child row, while the *START WITH* clause specifies where in the hierarchy the result set should begin.

Oracle supports the following types of *JOIN* syntax (refer to the [JOIN Subclause](#) section for more details):

```
FROM table1 {
  CROSS JOIN table2 |
  INNER JOIN table2 [ {ON join_condition |
    USING (column_list)} ] |
  LEFT [OUTER] JOIN table2 [ {ON join_condition
    | USING (column_list)} ] |
  NATURAL [LEFT [OUTER]] JOIN table2 |
  RIGHT [OUTER] JOIN table2 [ {ON join_condition
    | USING (column_list)} ] |
  NATURAL [RIGHT [OUTER]] JOIN table2
  FULL [OUTER] JOIN table2 }
```

[CROSS] JOIN

Retrieves all records of both *table1* and *table2*. This is syntactically the same as *FROM table1, table2* with no join conditions in the *WHERE* clause.

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* with join conditions in the *WHERE* clause is semantically equivalent to an inner join.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause containing all columns that are in common between the two joined tables. (Be careful if columns have the same name but not the same datatype or the same sort of values!) *LOB* columns cannot be referenced in a natural join. Referencing a *LOB* or collection column in a *NATURAL JOIN* will return an error.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all the records in a table, even when there are no counterparts in the joined table. For example:

```
SELECT j.job_id, e.lname
FROM jobs j
LEFT OUTER JOIN employee e ON j.job_id = e.job_id
ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. A right join is the same as a left join, except that the optional table is on the left.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

ON join_condition

Declares the condition(s) that join the result set of two tables together. This takes the form of declaring the columns in *table1* and *table2* that must match the join condition. When multiple columns must be compared, use the *AND* clause.

USING (column_list)

Acts as an alternative to the *ON* clause. Instead of describing the conditions of the join, simply provide a column name (or columns separated by commas) that appears in both tables. The column names must be identical in both tables and cannot be prefixed with a table name or alias. *USING* cannot be used on *LOB* columns of any type. The following two queries produce identical results. One is written with a *USING* clause and the other specifies join conditions using ANSI syntax:

```
SELECT column1
FROM foo
LEFT JOIN poo USING (column1, column2);
```

```
SELECT column1
FROM foo
LEFT JOIN poo ON foo.column1 = poo.column1
AND foo.column2 = poo.column2;
```

Note that older Oracle syntax for joins centered around the *WHERE* clause and outer joins were described using a (+)

marker. ANSI join syntax only became available with Oracle 9i Release 1. Therefore, you will often see existing code using the old syntax. For example, the following query is semantically equivalent to the earlier example of a left outer join.

```
SELECT j.job_id, e.lname
FROM jobs j, employee e
WHERE j.job_id = e.job_id (+)
ORDER BY d.job_id
```

This older syntax is sometimes problematic and more difficult to read. You are strongly advised to use the ANSI standard syntax.

Partitioned outer joins

Partitioned outer joins are useful for retrieving sparse data that might otherwise not be easily seen in a result set. (The ANSI standard describes partitioned outer joins, but Oracle is the first to support them.) For example, our **product** table keeps track of all products we produce, while the **manufacturing** table shows when we produce them. Since we're not continuously making every product at all times, the joined data between the two tables may be sparse at times:

```
SELECT manufacturing.time_id AS time, product_name AS name,
       quantity AS qty
FROM product
PARTITION BY (product_name)
RIGHT OUTER JOIN times ON (manufacturing.time_id =
       product.time_id)
WHERE manufacturing.time_id
       BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
       AND TO_DATE('06/10/05', 'DD/MM/YY')
ORDER BY 2, 1;
```

returns the following:

time	name	qty
01-OCT-05	flux capacitor	10
02-OCT-05	flux capacitor	
03-OCT-05	flux capacitor	
04-OCT-05	flux capacitor	
05-OCT-05	flux capacitor	
06-OCT-05	flux capacitor	10
06-OCT-05	flux capacitor	8
01-OCT-05	transmorgrifier	10
01-OCT-05	transmorgrifier	15
02-OCT-05	transmorgrifier	
03-OCT-05	transmorgrifier	
04-OCT-05	transmorgrifier	10

04-OCT-05 transmorgrifier 11

05-OCT-05 transmorgrifier

06-OCT-05 transmorgrifier

The example query and result set show that partitioned outer joins are useful for retrieving result sets that might otherwise be hard to query due to sparse data.

Flashback queries

Oracle 10g also supports flashback queries — queries that keep track of previous values of the result requested by the *SELECT* statement. In the following set of example code, we'll issue a regular query on a table, change the values in the table with an *UPDATE* statement, and then query the flashback version of the data. First, the regular query:

```
SELECT salary FROM employees  
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY  
-----  
3800
```

Now, we'll change the value in the **employees** table and query the table to confirm the current value:

```
UPDATE employees SET salary = 4000  
WHERE last_name = 'McCreary';  
SELECT salary FROM employees  
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY  
-----  
4000
```

Finally, we'll perform a flashback query to see what the **salary** value was in the past:

```
SELECT salary FROM employees  
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY)  
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY  
-----  
3800
```

If we wanted to be more elaborate, we could find out all of the values of **salary** for a given time period, say, the last

two days:

```
SELECT salary FROM employees

VERSIONS BETWEEN TIMESTAMP

    SYSTIMESTAMP - INTERVAL '1' MINUTE AND

    SYSTIMESTAMP - INTERVAL '2' DAY

WHERE last_name = 'McCreary';
```

The results are:

```
SALARY
-----
4000
3800
```

The MODEL clause

Oracle Database 10g features a powerful new clause, called *MODEL*, which enables spreadsheet-like result sets from a *SELECT* statement. The *MODEL* clause, in particular, is designed to alleviate the need for developers to extract data from the database and put it into a spreadsheet, like Microsoft Excel, for further manipulation. It creates a multidimensional array in which cells can be referenced by dimension values. For instance, you might dimension an array on product and time, specifying column values that you wish to access via combinations of those two dimensions. You can then write rules that are similar in concept to spreadsheet formulas, that are executed in order to change values in your model, or that create new values, and perhaps even new rows, in your model.

Syntactically, the *MODEL* clause appears after the *GROUP BY* and *HAVING* clauses and before the *ORDER BY* clause. The earlier syntax diagram for Oracle's *SELECT* statement shows the position of the clause, and the syntax details are presented here:

MODEL model_clause

model_clause ::= [options] [return] [reference_models] main_model

options ::= [{IGNORE|KEEP} NAV] [UNIQUE {DIMENSION|SINGLE REFERENCE}]

return ::= RETURN {UPDATED|ALL} ROWS

reference_models ::= reference_model [reference_model...]

reference_model ::= REFERENCE model_name ON (subquery) model_columns

[options]

model_columns ::= [partition [alias]]

DIMENSION BY (column[, column...])

MEASURES (column[, column...])

partition ::= PARTITION BY (column[, column...])

column ::= {column_name|expression} [AS alias]

main_model ::= [MAIN model_name] model_columns [options] rules

rules ::= [RULES [UPSERT|UPDATE] [{AUTOMATIC|SEQUENTIAL} ORDER]]

[ITERATE (count) [UNTIL (termination_condition)]]

(rule[, rule...])

rule ::= [UPSERT|UPDATE] cell_reference [ordering] = expression

cell_reference ::= measure[*multi_column_for_loop*][*dimension_indexes*]

multi_column_for_loop ::= FOR (*dimension*[, *dimension*...]) IN (*dimension_values*)

dimension_values ::= (*{subquery}dimension_value*[, *dimension_value*...])
dimension_value ::= (*literal*[, *literal*...])
dimension_indexes ::= *index*[, *index*...]
index ::= *{condition}expression**single_column_for_loop*
single_column_for_loop ::= FOR *dimension* *{in_for_options}like_for_options*
in_for_options ::= IN (*{subquery}literal*[, *literal*...])
like_for_options ::= [LIKE *pattern*] FROM *start_literal* TO *end_literal*
 {INCREMENT|DECREMENT} diff_literal
ordering ::= ORDER BY (*order_column*[, *order_column*...])
order_column ::= *{expression}alias* [ASC|DESC] [NULLS FIRST|NULLS_LAST]

The parameters of this *MODEL* clause are as follows:

{IGNORE|KEEP} NAV

Specifies whether null or absent values (*NAV*) are retained as nulls (*KEEP*), or whether they are replaced with suitable defaults (*IGNORE*): zero for numeric types, 1-Jan-2000 for date types, an empty string for character types, and null for anything else.

UNIQUE {DIMENSION|SINGLE REFERENCE}

Specifies the scope within which the database ensures that a given cell reference points to a unique data value. Use *DIMENSION* to require that each *possible* cell reference, whether on the left or right side of a rule, represent a single value. Use *SINGLE REFERENCE* to perform that check only for those cell references that appear on the right-hand side of a rule.

RETURN {UPDATED|ALL} ROWS

Specifies whether all rows are returned from model processing, or whether only updated row are returned.

reference_model

Defines a reference model on which you cannot perform calculations, but containing values that you can reference from within your main query.

model_name ON (*subquery*)

Specifies the name and rowsource for a reference model.

alias

Specifies an alias for a partition.

DIMENSION BY (column[, *column*...])

Specifies the dimensions for a model. Values from these columns represent the set of index values that are used to identify cells in the multidimensional addressing space.

MEASURES (*column*[, *column*...])

Specifies the values associated with each unique combination of dimensions (e.g., with each cell of the model).

PARTITION BY [(]*column*[, *column*...][]

Splits a model into independent partitions based on the columns given. You can not partition reference models.

main_model

Represents the model on which you perform work. Rows from your containing *SELECT* feed into this model, rules are applied, and the resulting rows are returned.

MAIN *model_name*

Begins the definition of the main model, and also gives that model a name.

RULES [*UPSERT*|*UPDATE*]

Specifies whether rules may both create new cells and update existing cells (*UPSERT*), or whether they much only update existing cells (*UPDATE*). If you want your model to be able to create new rows in your result set, specify *UPSERT*. The default is *UPSERT*. You can also control this behavior on a rule-by-rule basis; see *rule* in the syntax.

{*AUTOMATIC*|*SEQUENTIAL*} ORDER

Specifies whether the optimizer determines the order in which rules are evaluated (*AUTOMATIC*), or whether rules are evaluated in the order in which you list them (*SEQUENTIAL*). The default is *SEQUENTIAL*.

ITERATE (*count*)

Requests that entire set of rules be evaluated repeatedly, *count* times. The default is to evaluate the set of rules just once.

UNTIL (*termination_condition*)

Specifies a condition that, when met, causes iteration to cease. You must still specify a *count*, which serves as somewhat of a safeguard against infinite loops.

measure[...]

Is a reference to one of the measures listed in the *MEASURES* clause. When you reference a measure, the square-brackets are part of the syntax. You must specify all dimensions, either via a subquery or by listing them, and the specific value of the measure associated with those dimensions will be returned, or referenced.

multi_column_for_loop

A for-loop iterating over multiple dimensions at one time. The easiest way to picture this is to think of a subquery from which each row represents a specific combination of dimensions.

dimension_indexes

A list of values, whether from columns or expressions, that collectively identify a unique cell in the model.

single_column_for_loop

A for-loop iterating over one dimension.

IN ({*subquery*} *literal*[, *literal*...])

The source of values for a for-loop may be a subquery, or it may be a specific list of literal values.

LIKE *pattern*

Allows you to insert dimension values into a pattern. Use a percent-sign to mark the location at which you want dimension values to be inserted. For example, use `FOR x LIKE 'A%B' FROM 1 TO 3 INCREMENT 1` to generate values such as 'A1B', 'A2B', 'A3B'.

`FROM start_literal TO end_literal {INCREMENT|DECREMENT} diff_literal`

Defines the starting and ending for-loop values, and also the difference between each subsequent value as the loop iterates from start to end.

`ORDER BY (order_column [, order_column...])`

Imposes an order of evaluation with respect to the cells referenced from the left side of a rule. Use this clause if you want a rule to be applied to cells in order. Otherwise, you have no guarantee as to the order in which the rule is applied to the cells that it affects.

Following is a list of functions that have been designed specifically for use in the *MODEL* clause:

`CV()` or `CV(dimension_column)`

Returns the current value of a dimension column. May be used only on the right-hand side of an expression in a rule. When the `CV()` form is used, the dimension column is determined implicitly based on the function call's position in a list of dimension values.

`PRESENTNNV(measure[dimension, dimension...], not_null, was_null)`

Returns either *not_null* or *was_null*, depending on whether the specified measure was null when model processing began. This function may be used only from the right-hand side of a rule expression.

`PRESENTV(measure[dimension, dimension...], did_exist, didnt_exist)`

Returns either *did_exist* or *didnt_exist*, depending on whether the specified measure existed when model processing began. This function may be used only from the right-hand side of a rule expression. Be aware that whether a measure existed is a completely separate question from whether that measure was null.

ITERATION_NUMBER

Returns zero on the first iteration through the rules, 1 on the second iteration, and so forth. This is useful when you want to base rule calculations on the number of iterations.

The following example demonstrates that the *MODEL* clause gives a normal *SELECT* statement the ability to construct a multidimensional array as a result set and calculate inter-row and inter-array values interdependently. The newly calculated values are returned as part of the *SELECT* statement's result set:

```
ELECT SUBSTR(region,1,20) country, SUBSTR(product,1,15) product,
    year, sales
FROM sales_view
WHERE region IN ('USA','UK')
MODEL RETURN UPDATED ROWS
    PARTITION BY (region)
    DIMENSION BY (product, year)
    MEASURES (sale sales)
    RULES (
        sales['Bounce',2006] = sales['Bounce',2005] + sales['Bounce',2004],
        sales['Y Box', 2006] = sales['Y Box', 2005],
        sales['2_Products',2006] = sales['Bounce',2006]
        + sales['Y Box',2006] )
```

ORDER BY region, product, year;

In this example, a query against the **SALES_VIEW** materialized view returns the sum of sales over the course of a few years for the regions **USA** and **UK**. The *MODEL* clause then falls between the *WHERE* clause and the *ORDER BY* clause. Since **SALES_VIEW** currently holds data for the years 2004 and 2005, we provide it rules to calculate figures for the year 2006.

The subclause *RETURN UPDATED ROWS*, limits the result set to the rows that were created or updated by the query. Next, the example defines the logical divisions of the data using data elements from the materialized view and using the *PARTITION BY*, *DIMENSION BY*, and *MEASURES* subclauses. The *RULES* subclause then references individual measures of the model by referring to combinations of different dimension values much like a spreadsheet macro references worksheet cells with specific lookups and references to ranges of values.

PostgreSQL

PostgreSQL supports a straightforward implementation of the *SELECT* statement. It supports *JOIN* and subquery applications. PostgreSQL also allows the creation of new temporary or permanent tables using the *SELECT...INTO* syntax.

```
SELECT [ALL | DISTINCT [ON (select_item [,...]) ] ]  
  
[AS alias [(alias_list) ] [,...]  
  
[INTO [[TEMP]ORARY] [TABLE] new_table_name]  
  
[FROM [ONLY] table1[.*] [,... ]  
  
[ [join type]]JOIN table2 {[ON join_condition] |  
  [USING (column_list)]}  
  
[WHERE search_condition]  
  
[GROUP BY group_by_expression]  
  
[HAVING having_condition]  
  
[ORDER BY order_by_expression [{ASC | DESC  
  | USING operator][,... ] }]  
  
[FOR UPDATE [OF column [,...]]  
  
[LIMIT {count | ALL} ][OFFSET [number_of_records]] ]
```

where:

```
[ALL | DISTINCT [ON (select_item [,...]) ] ]
```

Supports the *ALL* and *DISTINCT* keywords of the ANSI SQL standard, where *ALL* (the default) returns all rows (including duplicates) and *DISTINCT* eliminates duplicate rows. In addition, *DISTINCT ON* eliminates duplicates on only one the specified *select_items*, not on all of the *select_items* in the query (example below).

select_item

Includes the standard elements of a *select_item* list supported by the ANSI SQL standard. In addition to the asterisk (*) shorthand to retrieve all rows, you can use *table_name.** to retrieve all rows from an individual table.

```
AS alias [(alias_list) ]
```

Creates an alias or a list of aliases for one or more columns (or tables in the *FROM* clause). *AS* is required for *select_item* aliases, but not for *FROM* table aliases. (Some other database platforms treat the *AS* as an option when declaring an alias.)

```
INTO [[TEMP]ORARY] [TABLE] new_table_name
```

Creates a new table from the result set of the query. Both *TEMP* and *TEMPORARY* are acceptable usages to create a temporary table that is automatically dropped at the end of the session. Otherwise, the command creates a permanent table. Permanent tables created with this statement must have new, unique names but temporary tables may have the same name as an existing table. If you create a temporary table with the same name as an existing permanent table, the temporary table is used to resolve all operations against that table name while in the same session as the one that created it. Other sessions will continue to see the existing permanent table.

FROM [*ONLY*] *table1* [,..]

Specifies one or more source tables where the data resides. (Be sure to specify a join condition or a theta *WHERE* clause so that you don't get a Cartesian product of all records in all tables.) PostgreSQL allows inheritance in child tables of declared parent tables. Use the *ONLY* keyword to suppress rows from the child tables of your source table. (You can turn off this default inheritance globally with the command *SET SQL_Inheritance TO OFF*.) PostgreSQL also supports nested table subqueries (see the [SUBQUERY Substatement](#) section). The *FROM* clause is not needed when used for computation:

```
SELECT 8 * 40;
```

PostgreSQL will also include an implicit *FROM* on *SELECT* statements that include schema-identified columns. For example, the following query is acceptable (though not recommended):

```
SELECT sales.stor_id WHERE sales.stor_id = '6380';
```

GROUP BY *group_by_expression*

Allows *group_by_expressions* that can be the column name or the ordinal number of the column (as determined by its position in the *select_item* list). An example illustrates this concept under the *ORDER BY* entry just below.

ORDER BY *order_by_expression*

Allows *order_by_expressions* that can be the column name, its alias, or the ordinal number of the column (as determined by its position in the *select_item* list.) For example, the following two queries are functionally identical:

```
SELECT stor_id, ord_date, qty AS quantity
```

```
FROM sales
```

```
ORDER BY stor_id, ord_date DESC, qty ASC;
```

```
SELECT stor_id, ord_date, qty
```

```
FROM sales
```

```
ORDER BY 1, 2 DESC, quantity;
```

In single table *SELECT* statements, you may also order by columns of the table that do not appear in the *select_item* list. For example:

```
SELECT *
```

```
FROM sales
```

```
ORDER BY stor_id, qty;
```

ASC and *DESC* are ANSI standard. If not specified, *ASC* is the default. PostgreSQL sorts NULL values as higher than any other value, causing NULL values to appear at the end of *ASC* sorts and at the beginning of *DESC* sorts.

FOR UPDATE OF *column* [,...] LIMIT {*count* | ALL}][OFFSET [*number_of_records*]]]

Limits the number of rows returned by the query to the maximum specified by the integer *count*. The optional *OFFSET* keyword tells PostgreSQL to skip *number_of_records* before starting to return rows. You should make sure you have an *ORDER BY* clause when using *LIMIT* or you may get an unexpected result set. (PostgreSQL 7.0 and greater treat *LIMIT/OFFSET* as a hint and use it to create optimal, possibly very different query plans.)

PostgreSQL supports a handy variation of the *DISTINCT* clause — *DISTINCT ON (select_item [,...])*. This variation allows you to pick and choose the exact columns that are considered for elimination of duplications. PostgreSQL chooses the result set in a manner much like it does for *ORDER BY*. You should include an *ORDER BY* clause so that there's no unpredictability as to which record is returned. For example:

```
SELECT DISTINCT ON (stor_id), ord_date, qty
FROM sales
ORDER BY stor_id, ord_date DESC;
```

The above query retrieves the most recent sales report for each store based on the most recent order date. However, there would be no way to predict what single record would have been returned without the *ORDER BY* clause.

PostgreSQL supports only these types of *JOIN* syntax (refer to the [JOIN Subclause](#) section for more details):

```
FROM table1 [,...] {
    CROSS JOIN table2 |
    [INNER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    LEFT [OUTER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    NATURAL [LEFT [OUTER]] JOIN table2 |
    RIGHT [OUTER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    NATURAL [RIGHT [OUTER]] JOIN table2
    FULL [OUTER] JOIN table2 }
```

where:

[CROSS] JOIN

Retrieves all records of both *table1* and *table2*. This is syntactically the same as *FROM table1, table2*. The results are a Cartesian product (usually a very bad thing!).

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* is semantically equivalent to an inner join without a join condition (i.e., a Cartesian product). When using *FROM table1, table2*, be sure include the join condition(s) in the *WHERE* clause. If you leave out a join type, PostgreSQL assumes you want an inner join.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause that names all columns that have the same name in both tables. (Be careful if columns have the same name but not the same datatype or the same sort of values!) *NATURAL* is mutually exclusive of a specified join condition via *ON* or *USING*.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted. You can use *LEFT OUTER JOIN* to find all

the records in a table even when there are no counterparts in the joined table. For example:

```
SELECT j.job_id, e.lname
FROM jobs j
LEFT OUTER JOIN employee e ON j.job_id = e.job_id
ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. Analogous to a *LEFT JOIN*.

FULL [OUTER] JOIN

Retrieves all records in the inner join in addition to all records that have no matching records on both the left and right sides.

ON join_condition

Declares the condition(s) that join the result set of two (or more) tables together. This takes the form of declaring the columns in *table1* and *table2* that must be equal. When multiple columns must be compared, use the *AND* clause.

SQL Server

SQL Server supports most of the basic elements of the ANSI *SELECT* statement, including all of the various join types. SQL Server offers several variations on the *SELECT* statement, including optimizer hints, the *INTO* clause, the *TOP* clause, *GROUP BY* variations, *COMPUTE*, and *WITH OPTIONS*.

```
SELECT {[ALL | DISTINCT] | [TOP number [PERCENT]
    [WITH TIES]]} select_item [AS alias]
[INTO new_table_name ]
[FROM {[rowset_function | table1 [,...]] [AS alias]]
[ [join_type]JOIN table2 {[ON join_condition] ]
[WHERE search_condition ]
[GROUP BY {grouping_column [,...]| ALL}] [ WITH { CUBE | ROLLUP } ]
[HAVING search_condition ]
[ORDER BY order_by_expression [ ASC | DESC ] ]
[COMPUTE {aggregation (expression)} [,...]]
    [BY expression [,...]] ]
[FOR {BROWSE | XML { RAW | AUTO | EXPLICIT}
    [, XMLDATA][, ELEMENTS][, BINARY base64] ]
[OPTION ( <hint> [,...]) ]
```

where:

TOP number [PERCENT][WITH TIES]

Specifies that only the *number* of rows are retrieved in the query result set. If *PERCENT* is specified, only the first *number* percent of the rows are retrieved. *WITH TIES* is used only for queries with an *ORDER BY* clause.

This variation specifies that additional rows are returned from the base result set using the same value in the *ORDER BY* clause, appearing as the last of the *TOP* rows.

INTO *new_table_name*

Creates a new table from the result set of the query. You can use this command to create temporary or permanent tables. (Refer to SQL Server's rules for creating temporary or permanent tables in the [CREATE/ALTER TABLE Statement](#) section.) The *SELECT . . . INTO* command quickly copies the rows and columns queried from other table(s) into a new table using a non-logged operation. Since it is not logged, it is not affected by *COMMIT* and *ROLLBACK* statements.

FROM {[*rowset_function* | *table1* [,...]]}

Supports the standard behavior of the ANSI SQL *FROM* clause, including nested table subqueries. In addition, SQL Server supports a set of extensions called *rowset_functions*. Rowset functions allow SQL Server to source data from special or external data sources such as XML streams, full-text search file structures (a special structure in SQL Server used to store things like MS Word documents and MS PowerPoint slide shows within the database), or external data sources (like an MS Excel spreadsheet).

Examples are shown below. (Please see the SQL Server documentation for the full description of the available *FROM* {[*rowset_function* | *table1* [,...]]} options). Among the many possibilities, SQL Server currently supports the following *rowset_functions*:

CONTAINSTABLE

Returns a derived table from a specified table that contains at least one full-text index *TEXT* or *NTEXT* column. The records derived are based upon either a precise, fuzzy, weighted-match, or proximity-match search. The derived table is then treated like any other *FROM* data source.

FREETEXTTABLE

Similar to *CONTAINSTABLE* except that records are derived based upon a meaning search of '*freetext_string*'. *FREETEXTTABLE* is more useful for ad hoc queries against full-text tables, but less accurate than *CONTAINSTABLE*.

OPENDATASOURCE

Provides a means of sourcing data external to SQL Server via OLE DB without declaring a linked server, such as an MS Excel spreadsheet or a Sybase Adaptive Server database table. It is intended for the occasional ad hoc query. If you frequently retrieve result sets from external data sources, then you should declare a linked server.

OPENQUERY

Executes a pass-through query against a linked server. It is an effective means of performing a nested table subquery against a data source that is external to SQL Server. The data source must first be declared as a linked server.

OPENROWSET

Executes a pass-through query against an external data source. It is similar to *OPENDATASOURCE*, except that *OPENDATASOURCE* only opens the data source; it does not actually pass through a *SELECT* statement. *OPENROWSET* is intended for occasional, ad hoc usage only.

OPENXML

Provides a query-able, table-like view to an XML string.

GROUP BY [ALL] (*group_by_expression*) [WITH {CUBE | ROLLUP}]

Supports the ANSI SQL standard with some variations. The first noticeable difference is in syntax. Where the ANSI standard clause is *GROUP BY* [{*CUBE* | *ROLLUP*}] (*grouping_column* [,...]), SQL Server uses *GROUP BY* [ALL] (*group_by_expression*) [WITH {*CUBE* | *ROLLUP*}].

GROUP BY ALL tells SQL Server to provide *group_by* categories even when the matching aggregation is NULL. Normally, SQL Server does not return a category whose aggregation is NULL. It must be used only in conjunction with a *WHERE*

clause. *WITH {CUBE | ROLLUP}* tells SQL Server to perform additional higher-level aggregates of the summary categories. In the simplest terms, *ROLLUP* produces subtotals for the categories, while *CUBE* produces cross-tabulated totals for the categories.



The *GROUPING* function can be used to help differentiate normally occurring NULLs from the NULLs generated by *ROLLUP* and *CUBE* behavior.)

This example groups **royalty** and aggregate **advance** amounts. The *GROUPING* function is applied to the **royalty** column.

USE pubs

```
SELECT royalty, SUM(advance) 'total advance',  
        GROUPING(royalty) 'grp'  
FROM titles  
GROUP BY royalty WITH ROLLUP
```

The result set shows two NULL values under **royalty**. The first NULL represents the group of null values from this column in the table. The second NULL is in the summary row added by the *ROLLUP* operation. The summary row shows the total **advance** amounts for all **royalty** groups and is indicated by 1 in the **grp** column.

Here is the result set:

royalty	total advance	grp
-----	-----	---
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

ORDER BY

Functions normally as specified by the ANSI SQL standard. However, you should remember that SQL Server allows a variety of *collations* that can affect how the result set is evaluated. Thus, under certain collations "SMITH" and "smith" might evaluate and sort differently. Also note that you cannot *ORDER BY* columns of *TEXT*, *NTEXT*, or *IMAGE* datatype.

COMPUTE {*aggregation (expression)*} [,...] [BY *expression* [,...]]]

Generates additional aggregations, usually totals, that appear at the end of the result set. *BY (expression)* adds in subtotals and control breaks to the result set. *COMPUTE* and *COMPUTE BY* can be used simultaneously in the same query. *COMPUTE BY* must be coupled with an *ORDER BY* clause, though the *expression* used by *COMPUTE BY* can be a subset of the *order_by_expression*. The aggregation may be any of these function calls: *AVG*, *COUNT*, *MAX*, *MIN*, *STDEV*, *STDEVP*, *VAR*, *VARP*, and *SUM*. Examples are shown later in this section.

COMPUTE, in any form, does not work with the *DISTINCT* keyword or with *TEXT*, *NTEXT*, or *IMAGE* datatypes.

FOR {*BROWSE* | *XML* { *RAW* | *AUTO* | *EXPLICIT* } [, *XMLDATA*][, *ELEMENTS*][, *BINARY BASE64*]]

FOR BROWSE is used to allow updates to data retrieved in a DB-Library browse mode cursor. (DB-Library is the

original access methodology for SQL Server and has since been supplanted by OLE DB in most applications.)
FOR BROWSE can only be used against tables with a unique index and a column with the *TIMESTAMP* datatype.
FOR BROWSE cannot be used in *UNION* statements or when a *HOLDLOCK* hint is active.

FOR XML

Used to extract the result set as an XML document to the SQL Server 2000 client only. You must further define the resulting XML document as either *RAW*, *AUTO*, or *EXPLICIT*. *RAW* transforms each returned row into a generic XML element with the *<row/>* element tag. *AUTO* transforms the results into a simple, nested XML tree. *EXPLICIT* transforms the resulting XML tree into an explicitly defined shape. However, the query must be written such that the desired nesting information is specified explicitly. You may optionally attach a few extra control features.

XMLDATA

Returns the schema appended to the XML document. *ELEMENTS* returns the columns as subelements instead of mapping them to XML attributes.

BINARY BASE6

Returns the binary data in base64-encoded format. (Binary is the default format for *AUTO* mode, but must be declared for *RAW* and *EXPLICIT*.) *FOR XML* cannot be used in any sort of subquery, in conjunction with a *COMPUTE* clause, *BROWSE* clause, in the definition of a view, in the result set of a user-defined function, or in a cursor. Aggregations and *GROUP BY* are mutually exclusive of *FOR XML AUTO*.

OPTION (*hint* [,...])

Replaces elements of the default query plan with your own. Because the optimizer usually picks the best query plan for any query, you are strongly discouraged from placing optimizer hints into your queries.

Here's an example of SQL Server's *SELECT...INTO* capability. This example creates a table called **non_mgr_employees** using *SELECT...INTO*. The table contains the **emp_id**, first name, and last name of each non-manager from the **employee** table, joined with their job description taken from the **jobs** table:

--QUERY

```
SELECT  e.emp_id, e.fname, e.lname,  
        SUBSTRING(j.job_desc,1,30) AS job_desc  
INTO    non_mgr_employee  
FROM    employee e  
JOIN    jobs AS j ON e.job_id = j.job_id  
WHERE   j.job_desc NOT LIKE '%MANAG%'  
ORDER BY 2,3,1
```

The newly created and loaded table **non_mgr_employee** now can be queried like any other table.



SELECT...INTO should be used only in development or non-production code because it is not logged or recoverable.

Many of SQL Server's extensions to the ANSI *SELECT* statement involve the *GROUP BY* clause. For example, the *GROUP BY ALL* clause causes the aggregation to include NULL valued results to be included when they would normally be excluded. The following two queries are essentially the same except for the *ALL* keyword, yet they produce very different result sets:

```
-- standard GROUP BY
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty <= 10
GROUP BY type
ORDER BY type
-- Results
type      price
-----
business  17.3100
mod_cook  19.9900
popular_comp 20.0000
psychology 13.5040
trad_cook 17.9700
```

Compared to:

```
-- Using GROUP BY ALL
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty = 10
GROUP BY ALL type
ORDER BY type
-- Results
type      price
-----
business  17.3100
mod_cook  19.9900
popular_comp 20.0000
psychology 13.5040
trad_cook 17.9700
UNDECIDED NULL
```

COMPUTE has a number of permutations that can impact the result set retrieved by the query. The following example shows the sum of book prices broken out by type of book and sorted by type and then price:

```
--Query
SELECT type, price
FROM titles
WHERE type IN ('business','psychology')
AND price > 10
```

ORDER BY type, price

COMPUTE SUM(price) BY type

--Results

type price

business 11.9500

business 19.9900

business 19.9900

sum

=====

51.9300

type price

psychology 10.9500

psychology 19.9900

psychology 21.5900

sum

=====

52.5300

The *COMPUTE* clause behaves differently if you do not include *BY*. The following query retrieves the grand total of prices and advances for books over \$16.00:

--Query

SELECT type, price, advance

FROM titles

WHERE price > \$16

COMPUTE SUM(price), SUM(advance)

--Result

type price advance

business 19.9900 5000.0000

business 19.9900 5000.0000

mod_cook 19.9900 .0000

popular_comp 22.9500 7000.0000

popular_comp 20.0000 8000.0000

psychology 21.5900 7000.0000

psychology 19.9900 2000.0000

trad_cook 20.9500 7000.0000

sum

```
=====
165.4500
      sum
=====
41000.0000
```

You could even use *COMPUTE BY* and *COMPUTE* in the same query to produce subtotals and grand totals. (For the sake of brevity, we'll show an example query, but not the result set.) In this example, we find the sum of prices and advances by type for business and psychology books over \$16.00:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
COMPUTE SUM(price), SUM(advance) BY type
COMPUTE SUM(price), SUM(advance)
```

Don't forget that you must include the *ORDER BY* clause with a *COMPUTE BY* clause! (You do not need an *ORDER BY* clause with a simple *COMPUTE* clause without the *BY* keyword.) There are many permutations you can perform in a single query — multiple *COMPUTE* and *COMPUTE BY* clauses, *GROUP BY* with a *COMPUTE* clause, and even with an *ORDER BY* statement. It's actually fun to tinker around with the different ways you can build queries using *COMPUTE* and *COMPUTE BY*. It's not theme park fun, but what'dya want? This is a programming book!

SQL Server also includes the *FOR XML* clause that converts the standard result set output into an XML document. This is very useful for web database applications. You can execute queries with *FOR XML* directly against the database or within a stored procedure. For example, we can retrieve one of our earlier example queries as an XML document:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
FOR XML AUTO
```

The results aren't particularly pretty, but they're very usable:

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
-----
<titles type="business" price="19.9900"
advance="5000.0000"/><titles type="business" price="19.9900" advance="5000.0000"/>
<titles type="psychology" price="19.9900" advance="2000.0000"/><titles
type="psychology" price="21.5900" advance="7000.000
```

If you wanted the XML schema and/or XML elements fully tagged in the output, you could simply append the *XMLDATA* and *ELEMENTS* keywords to the *FOR XML* clause. The query would look like this:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
FOR XML AUTO, XMLDATA, ELEMENTS
```

SQL Server also implements a number of other enhancements to support XML. For example, the *OPENXML* rowset function can be used to insert an XML document into a SQL Server table. SQL Server also includes system stored procedures that can help you prepare and manipulate XML documents.

SEE ALSO

JOIN
WHERE

SET Statement

The *SET* statement assigns a value to a runtime variable. The variables may be platform-specific system variables or user-defined variables.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

SET variable = value

Keywords

variable

Denotes a system or user-defined variable.

value

Denotes a string or numeric value appropriate to the system or user-defined variable.

Rules at a Glance

Variable values are set for the duration of the session. The value assigned to the variable must match the datatype of the variable. For example, you cannot assign a string value to a variable that is declared as with a numeric datatype. The actual command to create a variable varies from platform to platform. For example, DB2, Oracle, and SQL Server use the *DECLARE* statement to declare a variable name and datatype, but other platforms may use other means of

creating a variable.

The *value* assigned to a variable does not have to be a literal value. It may be a dynamic value that is derived from a subquery. For example, we assign the maximum employee ID to the **emp_id_var** variable in the following example:

```
DECLARE emp_id_var CHAR(5)

SET emp_id_var = (SELECT MAX(emp_id)

FROM employees WHERE type = 'F')
```

In this example, an employee type of 'F' indicates that the employee is a full-time, salaried employee.

Programming Tips and Gotchas

The *SET* statement is easily transportable between the database platforms. Only Oracle uses a consistently different scheme for assigning a value to a variable. In the following example, we declare a variable on SQL Server called **emp_id_var** and assign a value to it:

```
DECLARE emp_id_var CHAR(5)

SET emp_id_var = '67888'
```

Now, we will perform the same action on an Oracle server:

```
DECLARE emp_id_var CHAR(5);

emp_id_var := '67888';
```

DB2

DB2 supports the basic form of *SET*, assigning values to local variables, output parameters, or special registers. It allows a single *SET* statement to set multiple variable values at one time. It also assigns values to the columns of a base table in a trigger. The command cannot set both types of variables in a single statement.

```
SET variable = { value | NULL | DEFAULT } [...]
```

The DB2 syntax elements are as follows:

variable

Identifies the target variable. SQL variables must be declared before they are used. A variable may also identify a column in the base table of a trigger.

value

Describes the *value* of the variable in a datatype-appropriate format. When assigning values to columns in a trigger, you may also use references to *OLD* and *NEW* correlation names. For more information on this concept, refer to the DB2 platform specifics in the [CREATE/ALTER TRIGGER Statement](#) section.

NULL

Specifies a NULL value for columns or variables that allow NULL values.

DEFAULT

Assigns the default value to columns created with the *WITH DEFAULT* clause or the *IDENTITY* clause. This option assigns NULL to columns that allow NULL and are not defined with a default or identity clause.

To set a single variable value:

```
SET new_var.order_qty = 125;
```

When setting multiple values with a single statement, the number of variables on the left side of the equal sign must equal the number of values on the right side of the equal sign. To set multiple variables in a single statement:

```
SET new_var.order_qty = 125, new_var.discount = 4;
```

When using the *SET variable = SELECT...* variation, the values returned in the result set of the *SELECT* statement must exactly match the variables item for item in ordinal position and datatype. If the *SELECT* statement does not return any rows, then null values are assigned to the variables. You can also use the *SELECT INTO* statement to assign multiple variable values with a single statement.

MySQL

SET, as a keyword in MySQL, has several uses. First, *SET* is a MySQL datatype that allows multiple values each separated by a comma. In addition, *SET* may assign a value to a user variable. That latter use is described here, and the syntax for it is:

```
SET @variable = value [...]
```

When setting multiple values in a single statement, set each value separately with a comma between:

```
SET new_var.order_qty = 125, new_var.discount = 4;
```

In addition, MySQL allows the use of *SELECT* to assign values to variables in the same fashion as that described under the ANSI description of *SET*. However, there are some weaknesses when using the *SELECT* method for assigning values to variables. The primary problem is that values are not assigned immediately within the *SELECT* statement. Thus, for the following example:

```
SELECT (@new_var := row_id) AS a,  
       (@new_var + 3) AS b  
FROM table_name;
```

the **@new_var** variable will not possess the newly selected value of **row_id + 3**. It will retain the value from the beginning of the statement. For this reason, it is a best practice to assign only one value to a variable at a time.

Oracle

The *SET* clause is not supported as the method of assigning values to variables. Instead, user-defined variables are simply assigned a value using the assignment indicator *:=*. The basic syntax is shown here:

```
variable := value
```

PostgreSQL

The PostgreSQL command *SET* is used to set a runtime variable.

```
SET variable { TO | = } { value | DEFAULT }
```

The runtime variable may be set to a string literal designated by *value*. Using the *DEFAULT* keyword sets the runtime variable to its default value. PostgreSQL v7.2 supports these runtime *variables*:

CLIENT_ENCODING NAMES

Sets multibyte client encoding on PostgreSQL installations built with multibyte support

DATESTYLE

Sets the style used to display dates and times. Supported styles include:

ISO

Depicts date and time as YYYY-MM-DD HH:MM:SS (the default ISO 8601 style).

SQL

Depicts the Oracle/Ingres style of date and time, not the date and time style mandated by ANSI SQL.

Postgresql

Depicts the long-time date and time format in PostgreSQL, though no longer the default.

German

Depicts the date and time as dd.mm.yyyy.

You can further qualify the *SQL* and *Postgresql* styles using the keywords as **European**, **US**, and **NonEuropean**, which give dates in the forms dd/mm/yyyy, mm/dd/yyyy, and mm/dd/yyyy, respectively. For example: **SET DATESTYLE = SQL, European**.

SEED

Sets the internal random number generator seed. The value may be any floating-point number between 0 and 1 multiplied by $2^{31}-1$. This value can also be set via the PostgreSQL **setseed** function. For example:

```
SELECT setseed(value);
```

SERVER_ENCODING

Sets multibyte server encoding on servers built with multibyte support.

Here is an example for setting the date and time format to the Oracle and European style:

```
SET DATESTYLE TO sql, European;
```

SQL Server

SQL Server supports *SET* for variable assignments, provided the variables have previously been created with the *DECLARE* statement, and to define values for cursor variables in SQL Server. (SQL Server also uses the *SET* statement for a variety of other purposes, such as enabling or disabling session flags like *SET NOCOUNT ON*.) The platform-specific syntax is:

```
SET { { @variable = value }  
      | { @cursor_variable = { @cursor_variable | cursor_name }  
      | { CURSOR [ FORWARD_ONLY | SCROLL ]  
        [ STATIC | KEYSSET | DYNAMIC | FAST_FORWARD ]  
        [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
        [ TYPE_WARNING ]  
      FOR select_statement  
        [ FOR { READ ONLY | UPDATE [ OF column_name [,...] ] } }
```

] } } } }

The command does not support the *DEFAULT* keyword, but otherwise supports all the syntax of the ANSI command. The value for *server_name* must reference a connection named in a previous *CONNECT* statement, either as a literal or as a variable.

SET CONNECTION Statement

The *SET CONNECTION* statement allows users to switch between several open connections on one or more database servers.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

SET CONNECTION {DEFAULT | connection_name}

Keywords

connection_name

Names the connection of the current session. If the *connection_name* is different from the current session connection name, then the connection context is switched to the *connection_name*.

DEFAULT

Switches to the default connection from any other connection. This enables rapid switching to the default connection without knowing its name.

Description

This command does not end a connection. Instead, it switches from the current connection to the connection named (making it the current connection), or from the current connection to the default connection. When switching between connections, the old connection becomes dormant (without committing any changes), while the new connection becomes active.

Rules at a Glance

SET CONNECTION does not create a connection. It merely switches your connection context. Use the *CONNECT* command to create a new connection; use *DISCONNECT* to terminate a connection.

Programming Tips and Gotchas

The *SET CONNECTION* command is not frequently used, since many users connect programmatically via ODBC, JDBC, or some other connectivity method. However, on those platforms that support *SET CONNECTION*, the command can be very useful for rapidly changing connection properties without terminating any existing connections.

DB2

DB2 supports the following basic form of *SET CONNECTION*:

```
SET CONNECTION server_name
```

The DB2 command does not support the *DEFAULT* keyword, but is otherwise compatible with the ANSI command. The value for *server_name* must reference a connection created by a previous *CONNECT* statement.

Following is a fully embedded SQL program in DB2 that shows *CONNECT* and *SET CONNECTION*:

```
EXEC SQL CONNECT TO SQLNUTSHELL;  
  
/* Opens connection to SQLNUTSHELL. The connection will now  
   execute statements referencing objects at SQLNUTSHELL */  
  
EXEC SQL CONNECT TO DB2DEV;  
  
/* Opens connection to SQLNUTSHELL. The connection will now  
   execute statements referencing objects at DB2DEV */  
  
EXEC SQL SET CONNECTION SQLNUTSHELL;  
  
/* Now move back to the SQLNUTSHELL connection and execute  
   statements referencing objects at SQLNUTSHELL. The  
   connection to DB2DEV is still available, only dormant. */
```

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

SQL Server supports *SET CONNECTION*, but only in embedded SQL, not within its ad hoc querying tool, SQL Query Analyzer. Although SQL Server supports the full SQL2003 syntax within SQL embedded in other programs, such as a C++ program, it is not used very often. Most prefer the SQL Server specific command *USE* instead. The platform-specific syntax is:

```
SET CONNECTION connection_name
```

The command does not support the *DEFAULT* keyword, but is otherwise the same as the ANSI command. The value for *connection_name* must reference a connection named in a previous *CONNECT* statement, either as a literal or as a variable.

Here is a full TSQL program in SQL Server that shows *CONNECT*, *DISCONNECT*, and *SET CONNECTION*:

```
EXEC SQL CONNECT TO chicago.pubs AS chicago1 USER sa;  
  
EXEC SQL CONNECT TO new_york.pubs AS new_york1 USER read-only;  
  
-- opens connections to the servers named "chicago" and
```

```
"new_york"  
  
EXEC SQL SET CONNECTION chicago1;  
  
EXEC SQL SELECT name FROM employee INTO :name;  
  
-- sets the chicago1 connection as active and performs work  
    within that session  
  
EXEC SQL SET CONNECTION new_york1;  
  
EXEC SQL SELECT name FROM employee INTO :name;  
  
-- sets the new_york1 connection as active and performs work  
    within that session  
  
EXEC SQL DISCONNECT ALL;  
  
-- Terminates all sessions. You could alternately use two  
-- DISCONNECT commands, one for each named connection.
```

See Also

CONNECT
DISCONNECT

SET CONSTRAINT Statement

The *SET CONSTRAINT* statement defines, for the current transaction, whether a deferrable constraint checks after each DML statement or when the transaction is finally committed. If the session is not currently in an open transaction, the setting applies to the next transaction.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Not supported

SQL2003 Syntax

```
SET CONSTRAINT {constraint_name [,...] | ALL} {DEFERRED | IMMEDIATE}
```

Keywords

{*constraint_name* [,...] | ALL}

Names one or more deferrable constraints where the setting will apply. The keyword *ALL* sets the constraint mode for all deferrable constraints of the current transaction.

DEFERRED

Checks the conditions specified by the deferrable constraint when the transaction is committed, rather than when the DML statements are issued.

IMMEDIATE

Checks the conditions specified by the deferrable constraint immediately after each DML statement is issued, rather than when the transaction is committed.

Rules at a Glance

SET CONSTRAINT defines a value for the constraint mode of all deferrable constraints of the current transaction. If the session is not currently in a transaction, then the *SET CONSTRAINT* statement applies to the next transaction issued during the session.

The following example sets all deferrable constraints to be checked immediately following the issuance of each DML statement:

```
SET CONSTRAINT ALL IMMEDIATE;
```

The next example sets two constraints to defer their data modification until the transaction is committed:

```
SET CONSTRAINT scott.hr_job_title, scott.emp_bonus DEFERRED;
```

Programming Tips and Gotchas

Constraints, when they are defined, may be specified as *DEFERRABLE* or *NOT DEFERRABLE*. The *SET CONSTRAINT* statement will fail if it is issued against a specific constraint that was defined as *NOT DEFERRABLE*.

DB2

Not supported.

MySQL

Not supported.

Oracle

Oracle supports the ANSI standard exactly as described, except that *SET CONSTRAINT* is allowed, as well as *SET CONSTRAINTS*.

PostgreSQL

PostgreSQL supports the ANSI standard syntax exactly as it is described. Currently, PostgreSQL only allows *SET CONSTRAINT* against foreign key constraints, not check and unique constraints, which are always considered *IMMEDIATE*.

SQL Server

Not supported.

SET PATH Statement

The *SET PATH* statement changes the value of the *CURRENT PATH* setting to one or more schemas.

Platform	Command
DB2	Supported, with variations

MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

SET PATH schema_name [,...]

Keywords

schema_name [,...]

Defines one or more schemas as the current path.

Rules at a Glance

SET PATH defines one or more schemas used to qualify an unqualified routine name (that is, functions, procedures, and methods).

The following example sets the current path (i.e., schema name) for unqualified objects to **scott**:

SET PATH scott;

Then, whenever a routine is referenced during the current session, it will assume the **scott** schema if no schema is identified.

Programming Tips and Gotchas

When referencing multiple schema names, all the schemas must belong to the current database. (The schemas cannot be on a remote database.)

SET PATH does not apply the schema to unqualified objects like tables or views. It only applies to routines.

DB2

DB2 supports several variations on the ANSI standard:

SET {[CURRENT] PATH | CURRENT_PATH} =

{schema_name | SYSTEM PATH | USER | CURRENT PATH | CURRENT_PATH} [,...]

where:

SET {[CURRENT] PATH | CURRENT_PATH} =

Sets the path, with a slight variation in syntax from the ANSI standard. Note that the equal sign (=) is required.

schema_name

Defines one or more user-specified paths as the current path. The *schema_name* may be a literal, a host variable, or a string constant.

SYSTEM PATH

Defines the system schemas of *SYSIBM*, *SYSFUN*, and *SYSPROC* as the current path.

USER

Defines the value of the *USER* special register as the current path.

{CURRENT PATH | CURRENT_PATH}

Defines the value of the *CURRENT PATH* special register before the *SET PATH* statement was executed as the current path.

The following example sets the schema **scott** and the DB2 system schemas as the current path:

```
SET PATH = scott, SYSTEM PATH
```

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

SET SCHEMA

SET ROLE Statement

The *SET ROLE* statement enables and disables specific roles for the current session.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
SET ROLE {NONE | role_name}
```

Keywords

NONE

Assigns the *CURRENT ROLE* role to the current session.

role_name

Associates the set of privileges associated with the named role with the current session.

Rules at a Glance

When a user session is opened using the *CONNECT* statement, issuing the *SET ROLE* statement grants that session the privileges associated with a role. The *SET ROLE* command can be issued only outside of a transaction.

The value for *role_name* must reference a valid user role existing on the server. You may specify the role name either as a literal or through a variable.

Programming Tips and Gotchas

Sessions are created using the *CONNECT* statement, while roles are created using the *CREATE ROLE* statement.

Most database platforms offer some method of setting or changing the role used during a user session. *SET ROLE* is the ANSI standard approach to setting the role used during a user session, but it is not widely supported by the different database platforms. Check your platform to find an analogous command supported by your specific vendor.

DB2

Not supported. DB2 does not yet support roles.

MySQL

SET ROLE is not supported. An analogous method to control connection settings in MySQL is controlled in the `[client]` section of the `.my.cnf` configuration file in the home directory. For example:

```
[client]
host=server_name
user=user_name
password=client_password
```

To rapidly change between user roles, you can reassign host, user, and password connection properties by assigning new values to `MYSQL_HOST`, `USER` (for Windows only), and `MYSQL_PWD` (though `MYSQL_PWD` is insecure in that other users can view this file), respectively.

Oracle

When a user initiates a connection, Oracle explicitly assigns roles to the user. The role(s) under which the session is operating can be changed with the *SET ROLE* command, assuming the user has permissions to the role. Oracle uses the `MAX_ENABLED_ROLES` initialization parameter (in the `INIT.ORA` file) to control the maximum number of roles that can be enabled concurrently. The Oracle syntax is:

```
SET ROLE { role_name [IDENTIFIED BY password] [,...]
| [ALL [EXCEPT role_name [,...]]
| NONE } ;
```

Options for the *SET ROLE* command include:

role_name

Declares a valid role name already created within Oracle to which the user has been granted permissions. Any roles not specified are unavailable throughout the current session. You can enable multiple roles by providing a list; place a comma between each role listed.

IDENTIFIED BY password

If the *role_name* has a password, that password must be specified using this clause.

ALL

Enables all roles that are granted to the current user, including roles that are granted through other roles, but cannot be used with the *IDENTIFIED BY* clause.

EXCEPT

Specifies a list of roles to exclude from a *SET ROLE ALL* command.

NONE

Disables all roles, including the default role.

Roles with passwords may be accessed only through the statement *SET responsibility role_name IDENTIFIED BY password*. For example, to enable the specific roles **read_only** and **updater**, identified by the passwords **editor** and **red_marker**, respectively, for the current session:

SET ROLE

```
read_only IDENTIFIED BY editor,  
updater IDENTIFIED BY red_marker;
```

To enable all roles except the **read_write** role:

```
SET ROLE ALL EXCEPT read_write;
```

PostgreSQL

SET ROLE is not supported. However, the ANSI SQL command *SET SESSION AUTHORIZATION* is supported and can achieve somewhat similar results.

SQL Server

Not supported.

See Also

CONNECT
DISCONNECT
SET SESSION AUTHORIZATION

SET SCHEMA Statement

The *SET SCHEMA* statement changes the value of the *CURRENT SCHEMA* setting to a user-specified schemas.

Platform	Command
----------	---------

DB2	Supported, with variations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

SET SCHEMA schema_name [...]

Keywords

schema_name [...]

Defines one or more schema as the current path.

Rules at a Glance

SET SCHEMA defines a user-defined schema to use to qualify an unqualified object, such as a table or view.

The following example sets the current schema for unqualified objects to **scott**:

SET SCHEMA scott;

Then, whenever an object is referenced during the current session, it will assume the **scott** schema, if no schema is identified.

Programming Tips and Gotchas

The *SET SCHEMA* statement cannot assign a schema from a remote database as the *CURRENT SCHEMA*.

SET SCHEMA does not apply the schema to unqualified routines like functions, procedures, and methods. It only applies to database objects like tables and views.

DB2

DB2 supports several variations on the ANSI standard:

SET [CURRENT] SCHEMA = {schema_name | USER } [...]

where:

SET [CURRENT] SCHEMA =

Sets the path, with a slight variation in syntax from the ANSI standard. Note that the equal sign (=) is required.

schema_name

Defines a user-specified schema, of no more than 30 bytes in length, as the current schema. The *schema_name* may be a literal, a host variable, or a string constant.

USER

Defines the value of the USER special register as the current path.

The DB2 implementation is otherwise the same as the ANSI standard.

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

SET PATH

SET SESSION AUTHORIZATION Statement

The *SET SESSION AUTHORIZATION* statement sets the user identifier for the current session.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Not supported

SQL2003 Syntax

SET SESSION AUTHORIZATION *username*

Keywords

username

Sets the session user and the current user of the SQL session to the context of *username*, where *username* may be a literal, a parameter, or a host variable.

Rules at a Glance

This command allows you to switch between users and to run under their permissions.

Programming Tips and Gotchas

Some platforms allow you to use special shortcut keywords, like *SESSION USER* and *CURRENT USER*. *SESSION USER* and *CURRENT USER* are usually the same thing — the username of the currently active session provided by the client. However, *SESSION USER* and *CURRENT USER* can diverge in a session when *SETUID* functions and other similar mechanisms are invoked.

Superuser permissions are required to invoke this command, but you will still be able to switch back to the initial user session even if the current user session does not normally have permission to run *SET SESSION AUTHORIZATION*.

You might also wish to check the value of the *SESSION_USER* and *CURRENT_USER* functions with this SQL statement:

```
SELECT SESSION_USER, CURRENT_USER;
```

Normally, you should issue *SET SESSION AUTHORIZATION* before any transactions to set the session and current user values for all transactions that follow. It must be issued as the only command in its transaction batch.

DB2

Not supported. Similar functionality is achieved with the *CONNECT* statement or by logging off the server and reconnecting.

MySQL

Not supported. You must disconnect your session from MySQL, and then reconnect to utilize another set of user privileges.

Oracle

Not supported. Similar functionality is achieved using the *CONNECT* statement or by logging off the server and reconnecting.

PostgreSQL

PostgreSQL supports the ANSI standard for this command. The only difference, and it is a minor one, is that the ANSI standard does not allow this command during a transaction, while PostgreSQL does not care one way or the other.

SQL Server

Not supported. Similar functionality is achieved with the *CONNECT* statement or by logging off the server and reconnecting.

See Also

- CONNECT
- GRANT
- SESSION_USER

SET TIME ZONE Statement

The *SET TIME ZONE* statement changes the current session's time zone if it needs to be different from the default time zone.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Not supported

SQL2003 Syntax

```
SET TIME_ZONE {LOCAL | INTERVAL {+ | -}'00:00' [HOUR TO MINUTE]}
```

Keywords

LOCAL

Sets the current session's time zone to that of the local server.

INTERVAL

Sets an increase (with +) or decrease (with -) over the default time.

INTERVAL

Specifies the time zone offset from Coordinated Universal Time (UTC) in terms of hours and minutes.

HOUR TO MINUTE

Specifies the datatype of the *TIME_ZONE* value.

Rules at a Glance

This is a relatively simple command that sets the user session time zone to that of the server (*LOCAL*), or sets the time zone in relation to Coordinated Universal Time (*UTC*) (formerly Greenwich Mean Time or GMT). Thus, an *INTERVAL* of 2 would advance the time zone two hours greater than UTC, while an *INTERVAL* of -6 would reduce the time zone by six hours from UTC to the United States central time zone.

Programming Tips and Gotchas

Like most *SET* commands, *SET TIME_ZONE* can be executed only outside of an explicit transaction. In other words, you do not need to encapsulate the command within a *START* or *BEGIN TRAN* and a *COMMIT TRAN* statement.

DB2

Not supported.

MySQL

Not supported.

Oracle

In Oracle9i and higher, you can use the following *ALTER SESSION* command to set the session time zone:

```
ALTER SESSION
```

```
  SET TIME_ZONE = {'[+ | -] hh:mm'
```

```
    | LOCAL
```

```
    | DBTIMEZONE
```

```
    | 'region'}
```

Use *LOCAL* to revert back to your session's original default time zone. Use *DBTIMEZONE* to set your session time zone to be the database time zone. Use 'region' to specify a time zone region name such as EST or PST. Use an offset such as '-5:00' to specify your time zone in terms of an hour and minute displacement from UTC. A displacement of '-5:00' means that your time is five hours behind UTC time (e.g., 5:00 A.M. your time is 10:00 A.M. UTC time).

To retrieve a list of valid time zone region names, issue the following query:

```
SELECT tzone FROM v$timezone_names;
```

Both of the following commands set the time zone to Eastern Standard Time. The first command does this by specifying the appropriate displacement from UTC, while the second command specifies the time zone region name:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
```

```
ALTER SESSION SET TIME_ZONE = 'EST';
```

Oracle's time zone support is complex. Steven Feuerstein's *Oracle PL/SQL Programming* (O'Reilly), contains a good explanation of it in the chapter on datetime datatypes.

PostgreSQL

PostgreSQL allows a session's time value to be set to the server default by interchangeably using the *LOCAL* or *DEFAULT* clauses:

```
SET TIME ZONE {'timezone' | LOCAL | DEFAULT };
```

There are some variations from the ANSI standard:

'timezone'

Specifies a time zone by name. The possible values for time zone depends on the operating system. For example, the file */usr/share/zoneinfo* contains the database of time zones for Linux servers.

LOCAL / DEFAULT

Sets the current-session time values to those of the default time zone as determined by the local server.

For example, 'PST8PDT' is a valid time zone for California on Linux systems, while 'Europe/Rome' is a valid time zone for Italy on Linux and other systems. If you specify an invalid time zone, the command sets the time zone to UTC.

If increasing the time zone over UTC, the plus sign is optional. If an invalid time zone is specified, UTC is used as the time zone.

The following example sets the PostgreSQL time zone to Pacific Standard Time:

```
SET TIME ZONE 'PST8PDT';
```

Next, the time for the current session is returned to the server's default time zone:

```
SET TIME ZONE LOCAL;
```

SQL Server

Not supported.

SET TRANSACTION Statement

The *SET TRANSACTION* statement controls many characteristics of data modification, primarily the read/write characteristics and isolation level of a transaction.

Platform	Command
DB2	Not supported
MySQL	Supported, with variations
Oracle	Supported, with limitations
PostgreSQL	Supported
SQL Server	Supported, with variations

SQL2003 Syntax

```
SET [LOCAL] TRANSACTION [READ ONLY | READ WRITE]
    [ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
    REPEATABLE READ | SERIALIZABLE}]
    [DIAGNOSTIC SIZE inf]
```

Keywords

LOCAL

Changes transaction settings for the current session on the local server and only the local server. Otherwise, the transaction settings for the next transaction are changed, even if the transaction runs on a remote server.

READ ONLY

Sets the next upcoming transaction as a read-only transaction. Once the next transaction is complete, transaction behavior reverts to the default settings.

READ WRITE

Sets the next upcoming transaction so it may perform transactions that read and write data.

ISOLATION LEVEL

Sets the isolation level for the next transaction in the session.

READ COMMITTED

Allows a transaction to read rows written by other transactions only when they have been committed.

READ UNCOMMITTED

Allows a transaction to read rows which have been written, but not committed, by other transactions.

REPEATABLE READ

All sessions can see records that are committed before their first transaction was begun. Other open sessions can see or change only committed rows in the user's current session before their transactions were begun. Consequently, later transactions can add records that might then be visible to the transactions of earlier sessions, but the other sessions must query to see those records.

SERIALIZABLE

All sessions can see records that are committed before their transaction was begun. Other open sessions can see but cannot insert or update records within other user sessions until those transaction are completed. This is the most restrictive isolation level and the default for SQL2003.

DIAGNOSTIC SIZE int

Designates the specific number of error messages, *int*, to capture for a transaction. The *GET DIAGNOSTICS* statement retrieves these error messages.

Rules at a Glance

When issued, *SET TRANSACTION* sets the properties of the next upcoming transaction. Because of this, *SET TRANSACTION* is an interim statement that should be issued after one transaction completes and before the next transaction starts. (To begin a transaction and set its characteristics at the same time, use *START TRANSACTION*.) More than one option may be applied with this command but only one access mode, isolation level, and diagnostic size may be specified at a time.

The *isolation level* of a transaction specifies the degree of isolation a transaction has from other concurrently running sessions. The isolation level controls:

- Whether rows read and updated by your database session are available to other concurrently running database sessions.
- Whether the update, read, and write activity of other database sessions can affect your database session.

If you are unfamiliar with isolation levels, be sure to read the [SQL2003 Syntax](#) section under [SET TRANSACTION Statement](#), since the explanation for isolation levels is offered only in that section.

Programming Tips and Gotchas

The *ISOLATION LEVEL* clause controls a number of behaviors and anomalies in a transaction concerning concurrent transactions, particularly the following:

Dirty reads

Occur when a transaction reads the altered records of another transaction before the other transaction has completed. This allows a data modification to occur on a record that might not be committed to the database.

Nonrepeatable reads

Occur when one transaction reads a record while another modifies it. So if the first transaction attempts to reread the record, it can't find it.

Phantom records

Occur when a transaction A reads a group of records, but transaction B adds or changes the data so that more records satisfy the query issued by the transaction A. Thus, transaction A possibly reads in the records of transaction B as if they were committed to the database, when in fact the records from transaction B may still be rolled back. Since transaction A is reading records that are not yet permanent, these are called *phantom records*.

[Table 3-5](#) shows the impact of various isolation level settings on the anomalies just listed.

Table 3-5. Isolation level and anomaly impact

Isolation level	Dirty reads	Nonrepeatable reads	Phantom records
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	Not allowed	Allowed	Allowed

REPEATABLE READ	Not allowed	Not allowed	Allowed
SERIALIZABLE	Not allowed	Not allowed	Not allowed

DB2

Not supported. DB2 supports all isolation levels, but not through the *SET TRANSACTION* statement. Isolation levels are controlled instead through the program preparation process and in server-level settings issued by the database administrator.

Although DB2 supports the ANSI isolation levels, it uses different names for them. DB2 also allows users to set the isolation for specific statements, including *SELECT*, *DELETE*, *INSERT*, and *UPDATE*. DB2 uses the following names for the various isolation levels:

CS

Specifies cursor stability (serializable) as the isolation level.

NC

Specifies no commit as the isolation level. Not supported by DB2.

RR

Specifies repeatable read as the isolation level.

RS

Specifies read stability (read committed) as the isolation level.

UR

Specifies uncommitted read (read uncommitted) as the isolation level.

DB2 also supports two statements, *CHANGE ISOLATION LEVEL* and *SET CURRENT ISOLATION*, that change the way that DB2 isolates data from other processes while a database is being accessed.

MySQL

MySQL allows you to set the transaction isolation level for the next individual transaction, the whole session, or globally across the server, as follows:

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL

[READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ

| SERIALIZABLE]

By default, MySQL sets the isolation level for the transaction that immediately follows the statement.

GLOBAL

Sets the transaction isolation for all subsequent transactions across all user sessions or system threads.

SESSION

Sets the transaction isolation level for all subsequent transactions of the current session.

TRANSACTION ISOLATION LEVEL

Sets a specific transaction isolation level, as described in earlier in the [SET TRANSACTION Statement](#) section. When omitted, MySQL defaults to the *REPEATABLE READ* isolation level.

The *SUPER* privilege is required to set a *GLOBAL* transaction isolation level. You can also set the default isolation level via the *MYSQL* command line executable using the `-transaction-isolation=` switch. Following is an example that sets the all subsequent threads (both user and system threads) to a serializable transaction isolation level:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Oracle

Oracle allows you to set a transaction as read-only or read-write, set the transaction isolation level, or specify a specific rollback segment for your transactions.

```
SET TRANSACTION { [ READ ONLY | READ WRITE ]  
| [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]  
| [ USE ROLLBACK SEGMENT segment_name ]  
| NAME 'transaction_name'};
```

where:

READ ONLY

Sets the next transaction as read-only and serializable. This option is not available to the user **SYS**. The only statements permitted in read-only sessions are *SELECT*, *ALTER SESSION*, *ALTER SYSTEM*, *LOCK TABLE*, and *SET ROLE*.

READ WRITE

The default transaction style in Oracle. It allows transactions to read and write data.

READ COMMITTED

The default transaction isolation level in Oracle. The same as the ANSI standard.

SERIALIZABLE

Sets the transaction isolation level to the ANSI serializable level and requires the *COMPATIBLE* *init* parameter to be set to 7.3.0 or higher.

USE ROLLBACK SEGMENT *segment_name*

Sets the next read-write transaction to be written to a specific Oracle rollback segment identified by *segment_name*. Because it applies only to the current transaction, *USE ROLLBACK SEGMENT* should be the first statement in the transaction. This option is not compatible with the *READ ONLY* option. The rollback segment must already exist or else this statement will fail.

NAME

Assigns a name of 255 characters or less to the current transaction. This option is useful in distributed transaction processing environments, for two-phase commits, because it lets you easily identify which local transactions belong to a single distributed transaction.

The *USE ROLLBACK SEGMENT* variant can be useful for performance tuning by allowing you to direct long-running transactions to rollback segments large enough to hold them, while small transactions could be directed to small rollback segments that might be small enough to be retained in cache.

The *SET TRANSACTION* statement should be the first statement in any SQL batch, though Oracle treats it virtually the

same as the *START TRANSACTION* statement. So one could be substituted for the other.

In the following example, the query reports from a bi-weekly process on the Chicago server while avoiding impact from any other users who might be updating or inserting records:

```
SET TRANSACTION READ ONLY NAME 'chicago';

SELECT prod_id, ord_qty

FROM sales

WHERE stor_id = 5;
```

In another case, late-night batch processing might create a huge transaction that would overflow all but the rollback segment created to support that one transaction:

```
SET TRANSACTION USE ROLLBACK SEGMENT huge_tran_01;
```

PostgreSQL

SET TRANSACTION in PostgreSQL impacts the new transaction you are beginning only. Consequently, you may have to issue this statement before each new transaction.

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | SERIALIZABLE};
```

where:

READ COMMITTED

Sets the transaction isolation level to the ANSI level *READ COMMITTED*. This is the default.

SERIALIZABLE

Sets the transaction isolation level to the ANSI level *SERIALIZABLE*.

By default, PostgreSQL supports the *READ COMMITTED* transaction isolation level. You can set the default transaction isolation level for all transactions in the session by using either of the following commands:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
```

```
{ READ COMMITTED | SERIALIZABLE }
```

```
SET default_transaction_isolation =
```

```
{ 'read committed' | 'serializable' }
```

Of course, you can then override the isolation level of any subsequent transaction using the *SET TRANSACTION* statement.

For example, you can set the next transaction to the serializable transaction isolation level:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Alternately, you could set all the transactions in an entire session to serializable:

```
SET SESSION CHARACTERISTICS AS TRANSACTION
```

```
ISOLATION LEVEL SERIALIZABLE;
```

SQL Server

SET TRANSACTION in SQL Server sets the isolation level for an entire session. All queries that follow a *SET TRANSACTION* statement run under the isolation level set by the statement until it is otherwise changed.

SET TRANSACTION ISOLATION LEVEL

```
{ READ COMMITTED  
| READ UNCOMMITTED  
| REPEATABLE READ  
| SERIALIZABLE}
```

where:

READ COMMITTED

Sets the transaction isolation level to the ANSI level *READ COMMITTED*. This is the default.

READ UNCOMMITTED

Sets the transaction isolation level to the ANSI level *READ UNCOMMITTED*. This has the same effect as the *NOLOCK* optimizer hint.

REPEATABLE READ

Sets the transaction isolation level to the ANSI level *REPEATABLE READ*.

SERIALIZABLE

Sets the transaction isolation level to the ANSI level *SERIALIZABLE*. Similar results can be achieved in SQL Server using the *HOLDLOCK* optimizer hint.

For example, the following command lowers the transaction isolation level for all *SELECT* statements during the session from *READ COMMITTED* to *REPEATABLE READ*:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ  
GO
```

See Also

COMMIT
ROLLBACK

START TRANSACTION Statement

The *START TRANSACTION* statement allows you to perform all the functions of *SET TRANSACTION* while also initiating a new transaction.

Platform	Command
DB2	Not supported
MySQL	Supported, with limitations
Oracle	Not supported
PostgreSQL	Not supported; see <i>BEGIN TRAN</i> below

SQL Server	Not supported; see <i>BEGIN TRAN</i> below
------------	---

SQL2003 Syntax

START TRANSACTION [READ ONLY | READ WRITE]

[ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |

REPEATABLE READ | SERIALIZABLE}]

[DIAGNOSTIC SIZE *int*]

According to the ANSI standard, the only difference between *SET* and *START* is that *SET* is considered outside of the current transaction, while *START* is considered the beginning of a new transaction. Thus, *SET TRANSACTION* settings apply to the next transaction, while *START TRANSACTION* settings apply to the current transaction.

While only MySQL supports the *START TRANSACTION* statement, three of the vendors (MySQL, PostgreSQL, and SQL Server) support a similar command, *BEGIN [TRAN[SACTION]]* and its synonym *BEGIN [WORK]*. *BEGIN TRANSACTION* declares an explicit transaction, but it does not set isolation levels.

Rules at a Glance

The only significant rule of the *START TRANSACTION* statement is that you must use this statement to control the access mode, isolation level, or diagnostic size of the current transaction. Once a new transaction starts, you must either issue new values for the setting(s) or rely on the default.

Most database platforms allow you to implicitly control transactions, using what is commonly called *autocommit* mode. In autocommit mode, the database treats each statement as a transaction in and of itself, complete with an implicit *BEGIN TRAN* and *COMMIT TRAN* statement.

The alternative to autocommit mode is to manually control each transaction explicitly. Under explicit transaction control, you declare each new transaction with the *START TRANSACTION* statement. A new transaction may also start implicitly any time a transaction-initiating statement is issued, such as *INSERT*, *UPDATE*, *DELETE*, or *SELECT*. The transaction is not committed or rolled back until either *COMMIT* or *ROLLBACK* statements are explicitly issued.

Some platforms, namely DB2 and Oracle, do not support the explicit declaration of a new transaction using *START TRANSACTION*, but they do support explicitly committing, savepointing, and rolling back a transaction. Other platforms, such as MySQL, PostgreSQL, and SQL Server, allow you to both explicitly declare a transaction with *START TRANSACTION* and to explicitly commit, savepoint, and roll back the transaction.

Programming Tips and Gotchas

Many of the platforms discussed in this book run in autocommit mode by default. Therefore, it is a good rule of thumb to only use explicitly declared transactions if you intend to do so for all transactions in a session. In other words, do not mix implicitly declared transactions and explicitly declared transactions in a single session. Each transaction that is explicitly declared can only be made permanent with the *COMMIT* statement. Similarly, any transaction that fails or needs to be discarded must be explicitly undone with the *ROLLBACK* statement.



Be sure to issue *START* in a pair with either *COMMIT* or *ROLLBACK*. Otherwise, the DBMS may not complete the transaction(s) until it encounters *COMMIT* or *ROLLBACK*. Omitting timely *COMMITs* (or *ROLLBACKs*) could lead potentially to huge or poorly planned transactions.

It is a good idea to issue an explicit rollback or commit after one or a few statements because long-running transactions can lock up resources, thus preventing other users from accessing those resources. And don't forget to commit (or roll back)! Long-running or very large transaction batches can fill up the rollback segments or transaction logs of a database, even if those files are small.

DB2

Not supported. Transactions in DB2 are started implicitly.

MySQL

MySQL normally runs in autocommit mode, which means changes are automatically saved to disk when completed. If a change fails for any reason, it is automatically rolled back.

MySQL supports *START TRANSACTION*, but it is merely a synonym for *BEGIN*. You can suspend autocommit for one or several statements using the *BEGIN* syntax:

```
BEGIN [WORK]
```

where:

BEGIN

Marks the beginning of one or more transactions.

WORK

An optional keyword with no effect.

Issuing the following command can disable autocommit mode for all sessions and threads:

```
SET AUTOCOMMIT=0
```

Once you have disabled autocommit, the *COMMIT* statement is required to store any and every data modification to disk, and the *ROLLBACK* statement is required to undo changes made during a transaction. Disabling autocommit is only effective with "transaction-safe tables" such as **InnoDB** or **BDB** tables. Disabling autocommit on non-transaction safe tables has no effect — autocommit will still be enabled.



Earlier versions of MySQL use an update log. However, the update log does not support ANSI transactions unless the tables are defined as InnoDB or BDB.

Transactions are stored in the binary log in a single write operation when the *COMMIT* statement is issued. Here's an example:

```
BEGIN;
```

```
SELECT @A := SUM(salary)
```

```
FROM employee
```

```
WHERE type=1;
```

```
UPDATE payhistory SET summary=@A WHERE type=1;
```

```
COMMIT;
```

Rollbacks issued against non-transactional tables fail with the error **ER_WARNING_NOT_COMPLETE_ROLLBACK**, but transaction safe tables will be restored as expected.

Oracle

Not supported. Transactions in Oracle are started implicitly. Refer to *SET TRANSACTION* on the Oracle platform for more information about how Oracle controls individual transactions.

PostgreSQL

The PostgreSQL syntax is:

BEGIN [WORK | TRANSACTION]

where:

BEGIN

Marks the beginning of one or more transactions.

WORK

An optional keyword with no effect.

TRANSACTION

An optional keyword with no effect.

PostgreSQL normally runs in autocommit mode where each data modification statement or query is its own transaction. PostgreSQL applies an implicit *COMMIT* for a transaction that completes without an error. PostgreSQL also automatically issues a *ROLLBACK* statement at the end of a statement that fails. The *BEGIN* statement allows explicit *COMMIT* or *ROLLBACK* of a transaction, which may then consist of multiple statements.

Manually coded transactions are much faster in PostgreSQL than are autocommitted transactions. The *SET TRANSACTION ISOLATION LEVEL* should be set to *SERIALIZABLE* just after the *BEGIN* statement to bolster the transaction isolation. PostgreSQL allows many data-modification statements (*INSERT*, *UPDATE*, *DELETE*) within a *BEGIN* . . . *COMMIT* block. However, when the *COMMIT* command is issued, either all or none of the transaction takes place, depending on the success or failure of the command.



BEGIN has a separate usage on database platforms that support their own procedural language, namely Oracle and SQL Server. In these platforms, *BEGIN*, without the keyword *TRANSACTION*, is used to mark a new block of procedural code. For this reason, you are advised to include the *TRANSACTION* keyword for any transactions you write on PostgreSQL. Otherwise, you will face some complicated migration issues, should you ever move your code to Oracle or SQL Server.

Following is an example of *BEGIN TRANSACTION* in PostgreSQL:

```
BEGIN TRANSACTION;
```

```
    INSERT INTO jobs(job_id, job_desc, min_lvl, max_lvl)
```

```
    VALUES(15, 'Chief Operating Officer', 185, 135)
```

```
COMMIT;
```

SQL Server

Microsoft SQL Server supports the *BEGIN TRANSACTION* statement rather than the ANSI *START TRANSACTION* statement. It also supports a couple of extensions that facilitate transaction backup and recovery. The Microsoft SQL Server syntax is:

```
BEGIN TRAN[SACTION] [transaction_descriptor
```

```
    [WITH MARK [ 'log_descriptor' ] ] ]
```

where:

TRAN[SACTION]

Marks the beginning of a transaction. SQL Server allows either *TRAN* or *TRANSACTION*.

transaction_descriptor

An up to 32-character name used to identify a transaction, or it can be a variable string datatype (*CHAR*, *NCHAR*, *VARCHAR*, or *NVARCHAR*) variable up to 32 characters in size. When working with nested transactions, only name the outermost transaction.

WITH MARK *log_descriptor*

Tells SQL Server to place a mark of name *log_descriptor* in the transaction log. This allows SQL Server to restore a transaction log up to the specified *log_descriptor*. In a sense, this allows point in time recovery based on the name of the mark for database set to *FULL* recovery mode. *WITH MARK* must be used in conjunction with a named transaction.

When nesting transactions, only the outermost *BEGIN . . . COMMIT* or *BEGIN . . . ROLLBACK* pair should reference the transaction name (if it has one). In general, we recommend avoiding nested transactions.

Here is a SQL Server set of *INSERT* statements, all performed as a single transaction:

BEGIN TRANSACTION

```
INSERT INTO sales VALUES('7896','JR3435','Oct 28 2003',25,  
    'Net 60','BU7832')
```

```
INSERT INTO sales VALUES('7901','JR3435','Oct 28 2003',17,  
    'Net 30','BU7832')
```

```
INSERT INTO sales VALUES('7907','JR3435','Oct 28 2003',6,  
    'Net 15','BU7832')
```

COMMIT

GO

If, for any reason, any one of these *INSERT* statements had to wait for completion, they would all have to wait since they are treated as a single transaction.

See Also

COMMIT
ROLLBACK

SUBQUERY Substatement

A subquery is a nested query. Subqueries may appear in various places within a SQL statement.

Platform	Command
DB2	Supported
MySQL	Supported with limitations
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL supports the following types of subquery:

Scalar subqueries

Subqueries that retrieve a single value. These are the most widely supported type of subquery among the various database platforms.

Table subqueries

Subqueries that retrieve more than one values. This type of subquery retrieves more than a row of values.

Nested table subqueries

Subqueries that retrieve more than one column and more than one row.

Scalar and vector subqueries can, on some platforms, appear as part of the expression in a *SELECT* list of items, *WHERE* clause, and *HAVING* clause. Nested table subqueries tend to appear in the *FROM* clause of *SELECT* statements.

A *correlated subquery* is a subquery that is dependent upon a value in an outer query. Consequently, the inner query is executed once for every record retrieved in the outer query. Since subqueries can be nested many layers deep, a correlated subquery may reference any level in the main query higher than its own level.

Different rules govern the behavior of a subquery depending on which clause it appears in. The level of support amongst the database platforms also varies. Some platforms support subqueries in all clauses mentioned earlier (*SELECT*, *FROM*, *WHERE* and *HAVING*), while others support subqueries in only one or two of the clauses.

Subqueries are usually associated with the *SELECT* statement. Since subqueries may appear in the *WHERE* clause, they can be used in any SQL statement that supports a *WHERE* clause, including *SELECT*, *INSERT...SELECT*, *DELETE*, and *UPDATE* statements.

SQL2003 Syntax

Scalar, table, and nested table subqueries are represented in the generalized syntax below:

SELECT column1, column2, ... (scalar subquery)

FROM table1, ... (nested table subquery)

AS subquery_table_name]

WHERE foo = (scalar subquery)

OR foo IN (table subquery)

Correlated subqueries are more complex because the value of such subqueries is dependent on values retrieved in their main queries. For example:

SELECT column1

FROM table1 AS t1

WHERE foo IN

(SELECT value1

FROM table2 AS t2

WHERE t2.pk_identifier = t1.fk_identifier)

Note that the *IN* clause is for example purposes only. Any comparison operator may be used).

Keywords

scalar subquery

Includes a scalar subquery in the *SELECT* item list or in the *WHERE* or *HAVING* clause of a query.

nested table subquery

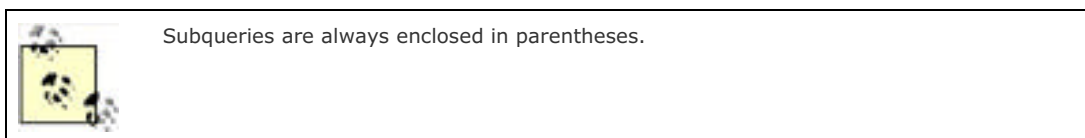
Includes a nested table subquery only in the *FROM* clause in conjunction with the *AS* clause.

table subquery

Includes a table subquery only in the *WHERE* clause with operators such as *IN*, *ANY*, *SOME*, *EXISTS*, or *ALL* that act upon multiple values. Table subqueries return one or more rows containing a single value.

Rules at a Glance

Subqueries allow you to return one or more values and nest them inside a *SELECT*, *INSERT*, *UPDATE*, or *DELETE* statement, or inside another subquery. Subqueries can be used wherever expressions are allowed. Subqueries also can often be replaced with a *JOIN* statement. Depending on the DBMS, subqueries may perform less quickly than joins.



Subqueries may appear in a *SELECT* clause with an item list containing at least one item, a *FROM* clause for referencing one or more valid tables or views, and in *WHERE* and *HAVING* clauses.

Scalar subqueries can return only a single value. Certain operators in a *WHERE* clause expect only one value, such as *=*, *<*, *>*, *>=*, *<=*, and *<>* (or *!=*). If a subquery returns more than one value against an operator that expects a single value, the entire query will fail. On the other hand, table subqueries may return multiple values, but are usable only with multivalued expressions like *[NOT] IN*, *ANY*, *ALL*, *SOME*, or *[NOT] EXISTS*.

Nested table subqueries may appear only in the *FROM* clause and should be aliased by the *AS* clause. The result set returned by the nested table subquery, sometimes called a *derived table*, offers similar functionality to a view (see *CREATE VIEW*). Every column returned in the derived table need not be used in the query, though they can all be acted upon by the outer query.

Correlated subqueries typically appear as a component of a *WHERE* or *HAVING* clause in the outer query (and, less commonly, in the *SELECT* item list) and are correlated through the *WHERE* clause of the inner query (that is, the subquery). (Correlated subqueries can also be used as nested table subqueries, though this is less common.) Be sure to include a *WHERE* clause in such a subquery that evaluates based on a correlating value from the outer query. (The example for a correlated query in the earlier ANSI syntax diagram illustrates this requirement.) It is important to specify a table alias, called a *correlation name*, using the *AS* clause or other alias shortcut for every table referenced in a correlated query, both in the outer and inner query. Correlation names avoid ambiguity and help the DBMS quickly resolve the tables involved in the query.

All ANSI-compliant subqueries comply with the following short list of rules:

- A subquery cannot include an *ORDER BY* clause.
- A subquery cannot be enclosed in an aggregate function. The following query is invalid: *SELECT foo FROM table1 WHERE sales >= AVG(SELECT column1 FROM sales_table...)*. You can get around this limitation by performing the aggregation in the subquery rather than in the outer query.

Programming Tips and Gotchas

For most all vendor platforms, subqueries should not reference large object datatypes (e.g., *CLOB* or *BLOB* on Oracle and *IMAGE* or *TEXT* on SQL Server), or array datatypes (such as *TABLE* or *CURSOR* on SQL Server).

The platforms all support subqueries, but not every vendor supports every type of subquery. [Table 3-6](#) tells you whether your platform supports each type of subquery.

Table 3-6. Platform-specific subquery support

Platform	DB2	MySQL	Oracle	PostgreSQL	SQL Server
Scalar subquery in SELECT item list	Y	Y	Y	X	Y
Scalar subquery in WHERE/HAVING clause	Y	Y	Y	Y	Y
Vector subquery in WHERE/HAVING clause	Y	Y	Y	Y	Y
Nested table in FROM clause	Y	Y	Y	Y	Y
Correlated subquery in WHERE/HAVING clause	Y	Y	Y	Y	Y

Subqueries are not relegated to *SELECT* statements only. They may also be used in *INSERT*, *UPDATE*, and *DELETE* statements that include a *WHERE* clause. Subqueries are often used for the following purposes:

- To identify the rows inserted into the target table using an *INSERT...SELECT* statement, *CREATE TABLE...SELECT* statement, or *SELECT...INTO*.
- To identify the rows of a view or materialized view in a *CREATE VIEW* statement.
- To identify value(s) assigned to existing rows using an *UPDATE* statement.
- To identify values for conditions in *WHERE* and *HAVING* clauses of *SELECT*, *UPDATE*, and *DELETE* statements.
- To build a view of a table(s) on the fly (i.e., nested table subqueries).

Examples

This section shows subquery examples that are equally valid on DB2, MySQL, Oracle, PostgreSQL, and SQL Server.

A simple scalar subquery is shown in the *SELECT* item list of the following query:

```
SELECT job, (SELECT AVG(salary) FROM employee) AS "Avg Sal"  
FROM employee
```

Nested table subqueries are functionally equivalent to querying a view. In the following, we query the education level and salary in a nested table subquery, and then perform aggregations on the values in the derived table in the outer query.

```
SELECT AVG(edlevel), AVG(salary)  
FROM (SELECT edlevel, salary  
      FROM employee) AS emprand  
GROUP BY edlevel
```



Remember that this query may fail, depending on the platform, without the *AS* clause to associate a name with the derived table.

The following query shows a standard table subquery in the *WHERE* clause expression. In this case, we want all project numbers for employees in the department 'A00':

```
SELECT projno  
FROM emp_act  
WHERE empno IN  
      (SELECT empno  
      FROM employee  
      WHERE workdept = 'A00')
```

The above subquery is executed only once for the outer query.

In the next example, we want to know the names of employees and their level of seniority. We get this result set through a correlated subquery:

```
SELECT firstname, lastname,  
  
       (SELECT COUNT(*)  
  
        FROM employee senior  
  
        WHERE employee.hiredate > senior.hiredate) as senioritytype  
  
FROM employee
```

Unlike the previous subquery, this subquery is executed one time for every row retrieved by the outer query. In a query like this, the total processing time could be very long, since the inner query may potentially execute many times for a single result set.

Correlated subqueries depend on values retrieved by the outer query before being able to complete the processing of the inner query. They are tricky to master, but offer unique programmatic capabilities. The following example returns information about orders where the quantity sold in each order is less than the average quantity in other sales for that title:

```
SELECT s1.ord_num, s1.title_id, s1.qty  
  
FROM sales AS s1  
  
WHERE s1.qty <  
  
       (SELECT AVG(s2.qty)  
  
        FROM sales AS s2  
  
        WHERE s2.title_id = s1.title_id)
```

For this example, you can accomplish the same functionality using a self join. However, there are situations in which a correlated subquery may be the only easy way to do what you need.

The next example shows how a correlated subquery might be used to update values in a table:

```
UPDATE course SET ends =  
  
       (SELECT min(c.begins) FROM course AS c  
  
        WHERE c.begins BETWEEN course.begins AND course.ends)  
  
WHERE EXISTS  
  
       (SELECT * FROM course AS c  
  
        WHERE c.begins BETWEEN course.begins AND course.ends)
```

Similarly, you can use a subquery to determine which rows to delete. This example uses a correlated subquery to delete rows from one table based on related rows in another table:

```
DELETE FROM course  
  
WHERE EXISTS  
  
       (SELECT * FROM course AS c  
  
        WHERE course.id > c.id  
  
        AND (course.begins BETWEEN c.begins  
  
            AND c.ends OR course.ends BETWEEN c.begins AND c.ends))
```

DB2

DB2 supports the ANSI standard subquery types. It allows scalar subqueries in the *SELECT* item list, nested table subqueries in the *FROM* clause, and scalar and vector subqueries in the expressions of *WHERE* and *HAVING* clauses.

DB2 allows correlated subqueries in the *SELECT* item list and in *WHERE* and *HAVING* clauses.

MySQL

MySQL supports nested table subqueries as select items and in the *WHERE* clause.

Oracle

Oracle supports ANSI standard subqueries, though it uses a different nomenclature. In Oracle, a nested table subquery that appears in the *FROM* clause is called an *inline view*. That makes plenty of sense because nested table subqueries are basically views built on the fly. Oracle calls a subquery that appears in the *WHERE* clause and *HAVING* clause of a query a *nested subquery*. Oracle allows correlated subqueries in the *SELECT* item list and in the *WHERE* and *HAVING* clauses.

PostgreSQL

PostgreSQL supports ANSI standard subqueries in the *FROM*, *WHERE*, and *HAVING* clauses. However, subqueries appearing in a *HAVING* clause cannot include *ORDER BY*, *FOR UPDATE*, and *LIMIT* clauses. PostgreSQL does not currently support subqueries in the *SELECT* item list.

SQL Server

SQL Server supports ANSI standard subqueries. Scalar subqueries can be used almost anywhere a standard expression is allowed. Subqueries in SQL Server cannot include the *COMPUTE* or *FOR BROWSE* clauses. They can include the *ORDER BY* clause if the *TOP* clause is also used.

See Also

DELETE
INSERT
SELECT
UPDATE
WHERE

TRUNCATE TABLE Statement

The *TRUNCATE TABLE* statement, a non-ANSI statement, irrevocably removes all rows from a table without logging the individual row deletes. It quickly erases all the records in a table without altering the table structure, taking little or no space in the redo logs or transaction logs. However, since a truncate operation is not logged, the *TRUNCATE TABLE* statement cannot be rolled back once it is issued.

Platform	Command
DB2	Not supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

Defacto Standard Syntax

Officially, *TRUNCATE TABLE* is not an ANSI standard command. However, it is a commonly supported statement standard that follows this format:

TRUNCATE TABLE *table_name*

Keywords

table_name

The name of any valid table within the current database or schema context.


Rules at a Glance

The *TRUNCATE TABLE* statement has the same effect on a table as a *DELETE* statement with no *WHERE* clause; both erase all rows in a given table. However, there are two important differences. *TRUNCATE TABLE* is faster and it is non-logged, meaning it cannot be rolled back if issued in error. Plus, *TRUNCATE TABLE* does not activate triggers, while the *DELETE* statement does.

This command should be issued manually. We strongly encourage you not to place it into automated scripts or production systems that contain irreplaceable data. It cannot be paired with transaction control statements such as *BEGIN TRAN* or *COMMIT*.

Programming Tips and Gotchas

Because the *TRUNCATE TABLE* statement is not logged, it is generally used only in development databases. Use it in production databases with caution!

	We strongly advise that you do not write <i>TRUNCATE TABLE</i> statements into the stored procedures or functions of a production application because most platforms do not log the operation nor can they recover an improperly issued <i>TRUNCATE</i> statement.
---	--

The *TRUNCATE TABLE* command will fail if another user has a lock on the table at the time the statement is issued. *TRUNCATE TABLE* does not activate triggers, but it will work when they are present. However, it won't work when foreign key constraints are in place on a given table.

DB2

As of Version 8, DB2 does not support the *TRUNCATE TABLE* statement.

MySQL

MySQL supports a basic format for the *TRUNCATE TABLE* statement starting in Version 3.23:

```
TRUNCATE TABLE name
```

MySQL achieves the results of a *TRUNCATE TABLE* command by dropping and recreating the affected table. Since MySQL stores each table in a file called *table_name.frm*, the *table_name.frm* file must exist in the directory containing the database files for the command to work properly.

For example, to remove all data from the **publishers** table:

```
TRUNCATE TABLE publishers
```

Oracle

Oracle allows a table or an indexed cluster (but not a hash cluster) to be truncated. Oracle's syntax is:

```
TRUNCATE { CLUSTER [owner.]cluster  
| TABLE [owner.]table [{PRESERVE | PURGE} MATERIALIZED VIEW LOG]}  
[{{DROP | REUSE} STORAGE}]
```

TRUNCATE TABLE was first introduced by Oracle, and other platforms soon added support for the statement. Oracle has added more features to this statement than are commonly implemented by other vendors. Oracle's syntax elements are as follows:

CLUSTER | TABLE

Specifies whether an index cluster or a table will be truncated. An index cluster is a physical construct that stores related rows of two tables next to each other physically on disk to speed up joins by lowering I/O. *SNAPSHOT LOG* and *STORAGE* options are not available when truncating an index cluster.

PRESERVE MATERIALIZED VIEW LOG

Maintains any snapshot logs when a master table is truncated.

PURGE MATERIALIZED VIEW LOG

Clears out any snapshot logs.

DROP STORAGE

Causes the disk space freed by the deleted rows to be deallocated.

REUSE STORAGE

Causes the space allocated to a table to remain allocated to that table, though empty.

For example:

```
TRUNCATE TABLE scott.authors
    PRESERVE SNAPSHOT LOG REUSE STORAGE
```

This example command erases all records in the table **scott.authors**. It also maintains the existing snapshot log and allows the table to keep and reuse the storage space already allocated to it.

PostgreSQL

PostgreSQL varies little from the defacto industry standard, but does add the optional *TABLE* keyword:

```
TRUNCATE [TABLE] name
```

The *TABLE* component of the command is optional. For example, the following command erases all the records in the **authors** table on a PostgreSQL database:

```
TRUNCATE authors
```

SQL Server

SQL Server supports the defacto industry standard.

See Also

DELETE

UNION Set Operator

The *UNION* set operator combines the result sets of two or more queries, showing all the rows returned by each of the queries as one, single result set.

UNION is in a class of keyword called a *set operator*. Other set operators include *INTERSECT* and *EXCEPT/MINUS*.

(*EXCEPT* and *MINUS* are functionally equivalent. *EXCEPT* is the ANSI standard.) All set operators are used to simultaneously manipulate the result sets of two or more queries; hence the term "set operators."

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

There are technically no limits to the number of queries that you may combine with the *UNION* statement. The general syntax is:

<SELECT *statement1*>

UNION [ALL | DISTINCT]

<SELECT *statement2*>

UNION [ALL | DISTINCT]

...

Keywords

UNION

Determines which result sets will be combined into a single result set. Duplicate rows are, by default, excluded.

ALL

Combines duplicate rows from all result sets.

DISTINCT

Eliminates duplicate rows from the final result set. Columns containing a NULL value are considered duplicates. (If neither *ALL* nor *DISTINCT* are used, *DISTINCT* behavior is the default.)

Rules at a Glance

There is only one significant rule to remember when using *UNION*: the order, number, and datatype of columns should be the same in all queries.

The datatypes do not have to be identical, but they should be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest of two (or more) compatible datatypes, and a query that unions three *CHAR* columns (*CHAR(5)*, *CHAR(10)*, and *CHAR(12)*) will display the results in the *CHAR(12)* format with extra space padded onto the smaller column results.

Programming Tips and Gotchas

Even though the ANSI standard calls for *INTERSECT* to take precedence over other set operators in a single statement, many platforms evaluate all set operators with equal precedence. Explicitly control the precedence of set operators using parentheses. Otherwise, the DBMS is likely to evaluate them from leftmost to rightmost expression.

Depending on the platform, *DISTINCT* can incur a significant performance cost, since it often adds a second pass through the results to winnow out duplicate records. *ALL* can be specified in any instance where no duplicate records are expected (or where duplicate records are OK) for faster results.

According to the ANSI standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. In order to avoid column and table ambiguity, be sure to alias each column for each table with the same respective alias. However, for column naming purposes, only the alias in the first query is used to each column in the *SELECT...UNION* query. For example:

```
SELECT au_lname AS "lastname", au_fname AS "firstname"

FROM authors

UNION

SELECT emp_lname AS "lastname", emp_fname AS "firstname"

FROM employees

ORDER BY lastname, firstname
```

Also, while your query may have compatible datatyped columns throughout the queries in the *UNION*, there may be some variation in behavior across the DBMS platforms, especially with regard to varying length of the columns. For example, if the **au_lname** column in the first query is markedly longer than the **emp_lname** column in the second query, each platform may apply different rules as to which length is used. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the columns across the tables have different names. In general, the column names of the first query are used.

DB2

DB2 supports the *UNION* and *UNION ALL* keywords of the ANSI standard, plus the *VALUES* clause:

```
<SELECT statement1>

UNION [ALL]

<SELECT statement2>

UNION [ALL]

...

[VALUES (expression1, expression2,
...)] [, (expression1, expression2, ...)] [,...]
```

where:

VALUES

Allows you to specify one or more sets of hand-coded values as records in the combined result set. Each set of hand-coded values must contain the exact same number of columns as the queries in the *UNION*. Separate rows of hand-coded values in the record set using commas.

Although *UNION DISTINCT* is not supported, *UNION* is the functional equivalent. The *CORRESPONDING* clause is not supported.

The datatypes *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK*, and structured types are not usable in a *UNION* (but are usable in *UNION ALL*).

If the result set has the same column name across tables, that name is used. If, however, the tables have different column names, DB2 generates a new column name. That column is then unusable in the *ORDER BY* clause or the *FOR UPDATE* clause.

If multiple set operations are used in a single query, those enclosed in parentheses are evaluated first. Then, the queries are evaluated in left-to-right order. However, all *INTERSECT* operations are performed before any *UNION* and *EXCEPT* operations. For example:

```
SELECT empno
FROM employee
WHERE workdept LIKE 'E%'
UNION
SELECT empno
FROM emp_act
WHERE projno IN ('IF1000', 'IF2000', 'AD3110')
UNION
VALUES ('AA0001'), ('AB0002'), ('AC0003')
```

The example above retrieves all employee IDs from the employee table that are in any department that starts with "E," as well as any employee IDs from the employee accounts (**emp_act**) table working in the projects **IF1000**, **IF2000**, and **AD3110**, and always includes employee IDs **AA0001**, **AB0002**, and **AC0003**.

MySQL

Not supported

Oracle

Oracle supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
UNION [ALL]
<SELECT statement2>
UNION [ALL]
...
```

Oracle does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent. Oracle does not support *UNION* or *UNION ALL* on queries under the following circumstances:

- Queries containing columns of *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatype.
- Queries containing a *FOR UPDATE* clause or a *TABLE* collection expression.

If the first query in the set operation contains any expressions in the select item list, then include the *AS* statement to associate an alias with the column. Also, only the last query in the set operation may contain an *ORDER BY* clause. For example, you could find out all unique store IDs without duplicates using this query:

```
SELECT stor_id FROM stores
UNION
SELECT stor_id FROM sales;
```

PostgreSQL

PostgreSQL supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
```

UNION [ALL]

<SELECT *statement2*>

UNION [ALL]

...

PostgreSQL does not support *UNION* or *UNION ALL* on queries with a *FOR UPDATE* clause. PostgreSQL does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *UNION* or *UNION ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the rightmost occurrence of *ORDER BY* or *LIMIT* will be assumed to apply to the entire set operation.

For example, we could find all authors and all employees whose last name starts with "P":

```
SELECT a.au_lname
```

```
FROM authors AS a
```

```
WHERE a.au_lname LIKE 'P%'
```

```
UNION
```

```
SELECT e.lname
```

```
FROM employee AS e
```

```
WHERE e.lname LIKE 'W%';
```

SQL Server

SQL Server supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

<SELECT *statement1*>

UNION [ALL]

<SELECT *statement2*>

UNION [ALL]

...

SQL Server does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent.

You can use *SELECT...INTO* with *UNION* or *UNION ALL*, but *INTO* may only appear in the first query of the union. Special keywords, such as *SELECT TOP* and *GROUP BY...WITH CUBE*, are usable with all queries in a union. However, be sure to include them with all queries in the union. If you use *SELECT TOP* or *GROUP BY...WITH CUBE* in only one query, the operation will fail.

Each query in a union must contain the same number of columns. The datatypes of the columns do not have to be identical, but they must implicitly convert. For example, mixing *VARCHAR* and *CHAR* columns is acceptable. When returning data, SQL Server uses the larger of the two columns when evaluating the size of the columns returned in the result set. Thus, if a *SELECT...UNION* statement has a *CHAR(5)* column and a *CHAR(10)* column, it will display the data of both columns as a *CHAR(10)* column. Numeric columns are converted to and displayed as the most precise datatype in the union.

For example, the following query unions the results of two independent queries that use *GROUP BY...WITH CUBE*:

```
SELECT ta.au_id, COUNT(ta.au_id)
```

```
FROM pubs..titleauthor AS ta
```

```
JOIN pubs..authors AS a ON a.au_id = ta.au_id
```

```
WHERE ta.au_id >= '722-51-5454'
```

```
GROUP BY ta.au_id WITH CUBE
UNION
SELECT ta.au_id, COUNT(ta.au_id)
FROM pubs..titleauthor AS ta
JOIN pubs..authors AS a ON a.au_id = ta.au_id
WHERE ta.au_id < '722-51-5454'
GROUP BY ta.au_id WITH CUBE
```

See also

EXCEPT
INTERSECT
MINUS
SELECT

UPDATE Statement

The *UPDATE* statement changes existing data in a table. Use great caution when issuing an *UPDATE* statement without a *WHERE* clause, since the statement then affects every row in the entire table.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
UPDATE [ONLY] {table_name | view_name}
SET {{column_name = { ARRAY [array_val [,...]] | DEFAULT |
NULL | scalar_expression},
column_name = { ARRAY | DEFAULT |
NULL | scalar_expression}
[,...]}
| ROW = row_expression }
[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

Keywords

ONLY

Restricts cascading of the updated values to any subtables of the target table or view. *ONLY* effects only typed (object-oriented) tables and views. If used with a nontyped table or view, it causes an error.

table_name | *view_name*

The target table or view of the update statement. You need appropriate permissions on the target according to the rules of the platform. Updates against views often have special rules. Generally, it is only advisable to perform an *UPDATE* against a view if the view is representative of a single table.

SET

Assigns a specific value to a column or row.

column_name

Used in conjunction with *SET*, as in *SET column1= 'foo'*. Allows you to set a new value for the specified column. You can update as many columns as you like in one statement, though you cannot update the same column more than once in a single statement.

ARRAY [*array_val* [,...]]

Assigns the column to an array value or to an empty array (using *ARRAY[]*). This behavior is not widely supported by DBMS platforms.

NULL

Sets the column to NULL.

DEFAULT

Sets the column to its default value as defined by a *DEFAULT* specification.

scalar_expression

Sets the column to any single value expression, such as a literal string or numeric value, a scalar function, or a scalar subquery.

ROW

Used in conjunction with *SET* is a mutually exclusive option to *SET column_name*, as in *SET ROW = ROW('foo','bar')*. This behavior is not widely supported by DBMS platforms.

row_expression

Used to set a value for each and every column in the table.

WHERE *search_condition*

Defines search criteria for the *UPDATE* statement using the *search_condition* to ensure that only the target rows are updated. Any legal *WHERE* clause is acceptable. Typically, these criteria are evaluated against each row of the table before the update is applied. If the search criteria is a subquery, the subquery is executed on each row — potentially a long-running process.

WHERE CURRENT OF *cursor_name*

Restricts the *UPDATE* statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

The *UPDATE* statement can be used to modify the value of one or more columns of a table or view at a time. Typically,

you will update the values in one or more rows at a time. The new values must be scalar (except in row expressions and arrays). That is, the new value must have a single, constant value at the time the transaction is executed, though it can be literal or derived from a function call or subquery.

A basic *UPDATE* statement without a *WHERE* clause looks like this:

```
UPDATE authors
SET contract = 0
```

Without a *WHERE* clause, all authors in the **authors** table have their contract status set to zero (meaning they don't have a contract any more). Similarly, values can be adjusted mathematically with an *UPDATE* statement:

```
UPDATE titles
SET price = price * 1.1
```

This *UPDATE* statement would increase all book prices by 10%.

You can set multiple values in one statement as here, where we set all author last name and first names to uppercase letters:

```
UPDATE authors SET au_lname = UPPER(au_lname),
    au_fname = UPPER(au_fname);
```

Adding a *WHERE* clause to an *UPDATE* statement allows records in the table to be modified selectively:

```
UPDATE titles
SET type = 'pers_comp',
    price = (price * 1.15)
WHERE type = 'popular_com';
```

The above query makes two changes to any record of the type '**popular_com**'. The command increases their price by 15% and alters their type to '**pers_comp**'.

You can also invoke functions or subqueries to derive the values used in an *UPDATE* statement. In some cases, you may wish to update the specific row that is being processed by a declared and open cursor. The following example shows both concepts:

```
UPDATE titles SET ytd_sales = (SELECT SUM(qty)
    FROM sales
    WHERE title_id = 'TC7777')
WHERE CURRENT OF title_cursor;
```

This query assumes that you have declared and opened a cursor named **title_cursor** and that it is processing titles with an ID of **TC7777**.

Sometimes you need to update values in a given table based on the values stored in another table. For example, if you need to update the publication date for all the titles written by a certain author, you might find the author and list of titles first through subqueries:

```
UPDATE titles
SET pubdate = 'Jan 01 2002'
WHERE title_id IN
    (SELECT title_id
    FROM titleauthor
```



```
WHERE au_id IN
(
  (SELECT au_id
   FROM authors
   WHERE au_lname = 'White'))
```

Programming Tips and Gotchas

The *UPDATE* statement alters values in existing records of a table or view. If you update a view, the view should include all necessary (*NOT NULL*) columns, or else the statement may fail. Columns with a *DEFAULT* value can usually be safely omitted.

The *SET ROW* clause is not widely supported among DBMS platforms. Avoid this clause. The *WHERE CURRENT OF* clause is more commonly implemented on DBMS platforms, but must be used in conjunction with a cursor. Make sure you understand the use and functionality of cursors before using this clause.

In an interactive user session, it is good practice to issue a *SELECT* statement using the same *WHERE* clause before issuing the actual *UPDATE* statement. This precaution enables you to check all rows in the result set before actually performing the *UPDATE*, helping ensure that you don't alter anything you don't mean to.

DB2

DB2 supports most of the standard structures found in the ANSI standard. It includes transaction control features using a transaction isolation level clause (the *WITH* clause), as well as typed table and view modification, but does *not* include the *SET ROW* clause.

```
UPDATE [ONLY] ( {table_name | view_name} ) AS alias
SET { {column_name1 = { DEFAULT | NULL | expression },
      column_name2 = { DEFAULT | NULL | expression }
      [,... ] } |
    { ( column_name1[..attribute_name],
        column_name2[..attribute_name][,...] ) =
      ( {DEFAULT | NULL | expression},
        {DEFAULT | NULL | expression}[,...] ) |
      SELECT statement }
[WHERE {search_conditions | CURRENT OF cursor_name}]
[WITH { RR | RS | CS | UR }]
```

where:

AS alias

Specifies an alias for the target table or view. This is useful for *UPDATE* statements that involve lengthy *WHERE* clauses or correlated subqueries.

..attribute_name

Specifies the *attribute_name* when updating a column that is a structured type.

WITH

Controls the transaction isolation level for the update statement (*RR* is repeatable read, *RS* is read stability, *CS* is cursor stability, and *UR* is uncommitted read). See *SET TRANSACTION* for details on DB2's transaction isolation levels.

The *UPDATE* statement must have only one value assignment per column. A column value can be set only once in an *UPDATE* statement. If any values of the update violate a constraint on the table or otherwise cause an error, then none of the columns are updated. When updating the value of an identity column to the current identity value, just use *DEFAULT* for that column. (DB2 fully supports updates on structure types, but they are neither commonly used nor quick and easy to explain. Refer to the DB2 documentation for more details on updating structured types.) DB2 allows parenthetical *SET* clauses where multiple target columns and values are assessed at once (see example later).

DB2 not only allows updates against standard tables and views, but also against typed (object-oriented) tables and views. (Refer to *CREATE TABLE* and *CREATE TYPE* for more details on these.) There are some special rules for updating against typed tables and views:

- The object ID column is not updateable.
- *..attribute_name* must be used in conjunction with a *column_name* that was previously defined as a user-defined structured type. The *attribute_name* must be an attribute of the structured type identified as *column_name*.
- Updating the column distribution statistics of a typed table requires specification of the subtable that first introduced the column.
- Multiple attribute assignments on the same structured type column occur in left-to-right order for a *SET* clause in parentheses, or else in the order specified in the *SET* clause.
- The same column may appear many times in a single *UPDATE* statement as long as each appearance sets a value for a specific *attribute_name*.

Updating the URL value of *DATALINK* columns is the same as deleting and then re-inserting them. Updating a *DATALINK* column with NULL is the same as deleting it. Updating the comment value of a *DATALINK* without relinking requires passing an empty string for the URL value.

Here are two examples that show how DB2 updates multiple values at once using the standard *SET* clause:

```
UPDATE employee
SET job=NULL, salary=0, bonus=0, comm=0
WHERE workdept = 'E21'
AND job <> 'MANAGER'
```

This statement could also be written using a parenthetical *SET* clause, as follows:

```
UPDATE employee
SET (job, salary, bonus, comm) = (NULL, 0, 0, 0)
WHERE workdept = 'E21'
AND job <> 'MANAGER'
```

Here is an example of a DB2 update against a base table on a type called *circles*. The *circles* attributes are owner, radius, X-axis center, and Y-axis center. Since the column **C** is defined as a type of **circle**, the *UPDATE* statement is changing the attributes of a single column:

```
UPDATE CIRCLES
SET (OWNER, C..RADIUS_LENGTH, C..X_CENTER, C..Y_CENTER) =
('tony', 5, 18, 22)
WHERE ID = 15
```

MySQL

MySQL supports the ANSI standard with a few variations, including the *LOW PRIORITY* clause, the *IGNORE* clause, and the *LIMIT* clause:

```
UPDATE [LOW PRIORITY] [IGNORE] table_name
SET column_name = {scalar_expression}
    [,...]
WHERE search_conditions
[ORDER BY column_name1 [{ASC | DESC}] [,...]]
[LIMIT integer]
```

where:

LOW PRIORITY

Tells MySQL to delay the execution of the *UPDATE* statement until no other client is reading from the table.

IGNORE

Tells MySQL to ignore any duplicate key errors generated by *PRIMARY KEY* and *UNIQUE* constraints. However, only records that don't generate such errors will be updated.

ORDER BY

Tells MySQL to update the rows in the specified order (ascending or descending) of the columns specified.

LIMIT integer

Restricts the *UPDATE* action to a specific number of rows as designated by the integer value.

MySQL supports *DEFAULT* value assignment in the *INSERT* statement, but not currently in the *UPDATE* statement.

The following *UPDATE* would increase all prices in the **titles** table by 1:

```
UPDATE titles SET price = price + 1;
```

The following example limits the *UPDATE* to the first 10 records encountered in the **titles** table according to **title_id**. Also, the value of **price** is updated twice. Those two updates of **price** are assessed from left to right. First, the price is doubled, then it is increased by 1:

```
UPDATE titles
SET price = price * 2,
    price = price + 1
ORDER BY title_id
LIMIT 10;
```

Oracle

The Oracle implementation of *UPDATE* allows updates against views, materialized views, subqueries, and tables in an allowable schema:

```
UPDATE [ONLY]
{ [schema}.{view_name | materialized_view_name |
    table_name}
    [ @database_link ] [alias]
```

```
{[PARTITION (partition_name) ] |  
 [SUBPARTITION (subpartition_name)]}  
 | subquery [WITH {[READ ONLY] | [CHECK OPTION [CONSTRAINT constraint_name] ]  
 | [TABLE (collection_expression_name) [ ( + ) ] ] }  
SET {column_name1 [,...] = {expression [,...] | subquery} |  
  VALUE [(alias)] = { value | (subquery)},  
 {column_name1 [,...] = {expression [,...] | subquery} |  
  VALUE [(alias)] = { value | (subquery)},  
 [,...]  
WHERE search_conditions | CURRENT OF cursor_name}  
RETURNING expression [,...] INTO variable [,...];
```

Syntax elements for Oracle's version of *UPDATE* are:

ONLY

Tells Oracle not to update rows from any subviews. Applies only to views that belong to a hierarchy.

materialized_view_name

Applies the update statement to a pre-existing snapshot.

@database_link

Applies the update statement to a view or table on a remote server made available through a pre-existing database link

alias

Specifies an alias for the table being updated. Oracle allows table aliases, but does not support the *AS* keyword with *UPDATE* statements. Aliases can also be supplied for the *VALUE* clause of an *UPDATE* statement.

PARTITION

Applies the update to a named partition, rather than to an entire table. You are not required to name a partition when updating a partitioned table, but doing so can, in many cases, help reduce the complexity of your *WHERE* clause.

SUBPARTITION

Applies the update to a named subpartition, rather than to an entire table.

WITH READ ONLY

Specifies that the *subquery* cannot be updated. In some cases, an *UPDATE* transaction may actually alter the records used in a subquery. The *WITH READ ONLY* clause prevents this from happening.

WITH CHECK OPTION

Tells Oracle to abort any changes to the updated table that would not appear in the result set of the subquery.

CONSTRAINT

Tells Oracle to further restrict changes based upon a specific constraint.

SET VALUE

Allows you to set the entire row value for any *TABLE* datatype. The *SET VALUE* clause is very similar to the ANSI *SET ROW* clause. You can also specify a subquery that retrieves all values needed by the *SET VALUE* clause.

RETURNING

Retrieves the rows affected by the command, whereas *UPDATE* normally only shows the number of rows updated. When used for a single-row update, the values of the row can be stored in PL/SQL variables and bind variables. When used for a multirow delete, the values of the rows are stored in bind arrays.

INTO

Indicates the variables into which the values brought back by the *RETURNING* clause should be stored.

Oracle has some important rules about issuing *UPDATE* statements:

- *UPDATE* statements are *not* allowed against tables or the base tables of views containing a domain index with a status of *FAILED* or *IN PROGRESS*. *UPDATE* statements are also not allowed against any table having an index partition with a status of *UNUSABLE*. (You can avoid the index partition problem by setting the *SKIP_UNUSABLE_INDEXES* session parameter to *TRUE*.)
- Updates against views can be problematic if a view's defining query contains a join, an aggregate function, a set operator, the *DISTINCT* keyword, an *ORDER BY* clause, a *GROUP BY* clause, a *CONNECT BY* clause, a *START WITH* clause, an analytical function, a collection expression, or a subquery. (Under these circumstances, you can use an *INSTEAD OF* trigger to update the tables underlying a view.)

The following code snippets show some examples of the extensions that Oracle offers in the *UPDATE* statement. To begin with, assume that the **sales** table has grown into a large partitioned table. You can update a specific partition using the *PARTITION* clause, for example:

```
UPDATE sales PARTITION (sales_yr_2004) s
```

```
    SET s.payterms = 'Net 60'
```

```
WHERE qty > 100;
```

You can also use the *SET VALUE* clause to supply a new value for each column in the row identified by the *WHERE* clause:

```
UPDATE big_sales bs
```

```
    SET VALUE(bs) = (SELECT VALUE(s) FROM sales s
```

```
        WHERE bs.title_id = s.title_id)
```

```
        AND bs.stor_id = s.stor_id)
```

```
WHERE bs.title_id = 'TC7777';
```

You can return values for review after an update by using the *RETURNING* clause. The following example updates one row, placing the updated values into PL/SQL variables:

```
UPDATE employee
```

```
    SET job_id = 13, job_level = 140
```

```
WHERE last_name = 'Josephs'
```

```
RETURNING last_name, job_id
```

```
INTO :var1, :var2;
```

PostgreSQL

PostgreSQL supports the ANSI standard *UPDATE* syntax with a couple of small variations. PostgreSQL supports the addition of a *FROM* clause. PostgreSQL's other variations include no support for the *ARRAY* keyword, but support for an array functionality. PostgreSQL's *UPDATE* syntax is:

```
UPDATE [ONLY] {table_name | view_name}

SET column_name1 = {DEFAULT | NULL | scalar_expression },
    column_name2 = {DEFAULT | NULL | scalar_expression }

[,...]

[FROM {table1 [AS alias], table2 [AS alias] [...]}]

[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

Most syntax elements are the same as in the ANSI standard. The only non-standard element is:

FROM

Provides the ability to build highly selective join-based criteria when determining which rows to update. *FROM* is not needed when only one table, the target table, is used to determine the rows updated.

In the following example, we want to update **job_lvl** for employees who have a **job_id** of 12 and a **min_lvl** of 25. We could do this with a large, complex subquery, or we could use the *FROM* clause to enable us to build a join:

```
UPDATE employee

SET job_lvl = 80

FROM employee AS e, jobs AS j

WHERE e.job_id = j.job_id

AND j.job_id = 12

AND j.min_lvl = 25;
```

All other aspects of the ANSI standard are fully supported.

SQL Server

SQL Server supports most of the basic components of the ANSI *UPDATE* statement, but it does not support the *ONLY* and *ARRAY* keywords, nor does it support array update functionality. SQL Server has extended the capabilities of *UPDATE* by adding table hints using the *WITH* clause, query hints using the *OPTION* clause, as well as more robust variable handling, as follows:

```
UPDATE {table_name | view_name | rowset}

[WITH (hint1, hint2 [...])]

SET {column_name = {DEFAULT | NULL | scalar_expression }
    | variable_name = scalar_expression
    | variable_name = column_name = scalar_expression } [...]

[FROM {table1 | view1 | nested_table1 | rowset1} [...]]

[AS alias]

[JOIN {table2 [...]}]

WHERE {conditions | CURRENT OF [GLOBAL] cursor_name}
```

[OPTION (*hint1*, *hint2* [,...])]

SQL Server's *UPDATE* syntax elements are as follows:

WITH *hint*

Allows the use of table hints to override the default behavior of the query optimizer. Since the query optimizer is quite good at choosing query plans, use hints only with a deep understanding of the tables, indexes, and data affected by the operation. Without this understanding, including hints could result in a decrease rather than an increase in performance.

variable_name

SQL Server variables must be declared prior to the *UPDATE* statement in the form of *DECLARE @variable*. The construct *SET @variable = column1 = expression1* sets a variable to the final value of an updated column, whereas *SET @variable = column1, column1 = expression* sets the variable to the value of the column before execution of the *UPDATE* statement.

FROM

Provides the ability to build highly selective join-based criteria when determining which rows to update. *FROM* is not needed when only one table, the target table, is used to determine the rows updated. SQL Server rowset functions are described in the [SELECT Statement](#) section.

AS *alias*

Allows you to assign an easy-to-use alias to the table, view, nested table subquery, or rowset function.

JOIN

Provides the ability to use ANSI standard syntax for joined tables, in conjunction with the *FROM* clause.

GLOBAL

A slight variation on the ANSI standard *WHERE CURRENT OF* clause. The clause *WHERE CURRENT OF cursor_name*, when used in combination with a cursor, tells SQL Server to update only the single record where the cursor is currently positioned. The cursor is assumed to be a local cursor, but can be designated a global cursor by using the keyword *GLOBAL*.

OPTION *hint*

Allows the use of query hints to override the default behavior of the query optimizer. Like the *WITH* clause, which uses table hints, use query hints only with a deep understanding of the tables, indexes, and data affected by the operation. Without this understanding, including hints could result in a decrease rather than an increase in performance.

The primary extension to the ANSI standard that Microsoft SQL Server offers in an *UPDATE* statement is a *FROM* clause. This *FROM* clause allows the use of the *JOIN* statement to make it especially easy to update rows in the target table by correlating rows declared in the *FROM* clause to the rows updated by the *UPDATE table_name* component of the statement. The following example shows an update using the ANSI style and a rather cumbersome subquery, followed by an update using SQL Server's *FROM* clause extension, to update the result of a table join. Both statements accomplish the same work, but in very different ways:

-- ANSI style

UPDATE titles

SET pubdate = GETDATE()

WHERE title_id IN

(SELECT title_id

FROM titleauthor

```
WHERE au_id IN
  (SELECT au_id
   FROM authors
   WHERE au_lname = 'White'))
-- Microsoft Transact-SQL style
UPDATE titles
SET pubdate = GETDATE( )
FROM authors AS a
JOIN titleauthor AS t2 ON a.au_id = t2.au_id
WHERE t2.title_id = titles.title_id
AND a.au_lname = 'White'
```

Performing this update using the Transact-SQL style is simply a matter of joining two tables — **authors** and **titleauthor** — to the **titles** table. To perform the same operation using ANSI-compliant code, the **au_id** in **author** must first be found and passed up to the **titleauthors** table, and then the **title_id** must be identified and passed up to the main update statement.

The following example updates the **state** column for the first 10 authors from the **authors** table:

```
UPDATE authors
SET state = 'ZZ'
FROM (SELECT TOP 10 * FROM authors ORDER BY au_lname) AS t1
WHERE authors.au_id = t1.au_id
```

The important thing to note about this example is that it is normally difficult to update the first *n* records in an *UPDATE* statement unless there is some explicit row sequence you can identify in the *WHERE* clause. However, the nested table subquery in the *FROM* clause uses a *TOP* keyword to return on the first 10 records, thereby saving a lot of added programming that would otherwise be required.

See Also

```
DECLARE
SELECT
WHERE
```

WHERE Clause

The *WHERE* clause sets the search criteria for an operation such as *SELECT*, *UPDATE*, or *DELETE*. Any records in the target table(s) that do not meet the search criteria are excluded from the operation. The search conditions may include many variations such as calculations, Boolean operators, and SQL predicates (for example, *LIKE* or *BETWEEN*).

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

{ WHERE *search_criteria* | WHERE CURRENT OF *cursor_name* }

Keywords

WHERE *search_criteria*

Defines search criteria for the statement to ensure that only the target rows are affected.

WHERE CURRENT OF *cursor_name*

Restricts the operation of the statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

WHERE clauses are found in *SELECT* statements, *DELETE* statements, *INSERT...SELECT* statements, *UPDATE* statements, and any statement that might have a query or subquery such as *DECLARE*, *CREATE TABLE*, *CREATE VIEW*, and so forth. The search conditions, all of which are described in their own entries elsewhere in the book, can include:

All records (=ALL, >ALL, <= ALL, SOME/ANY)

For example, to see publishers who live in the same city as their authors:

```
SELECT pub_name
FROM publishers
WHERE city = SOME (SELECT city FROM authors);
```

Combinations (AND, OR, and NOT) and evaluation hierarchy

For example, to see all authors with sales in quantities greater than or equal to 75 units, or secondary authors with a royalty of greater than or equal to 60:

```
SELECT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE ta.title_id IN (SELECT title_id FROM sales
                     WHERE qty >= 75)
   OR (a.au_id IN (SELECT au_id FROM titleauthor
                 WHERE royaltyper >= 60)
      AND a.au_id IN (SELECT au_id FROM titleauthor
                    WHERE au_ord = 2));
```

See [Programming Tips and Gotchas](#) next, for more information.

Comparison operators (such as =, <, >, <=, and >=).

For example, to see author's last and first name from the **authors** table for authors who don't have a contract (i.e., contract value of zero):

```
SELECT au_lname, au_fname
FROM authors
WHERE contract = 0;
```

Lists (IN and NOT IN).

For example, to see all authors who do not yet have a title in the **titleauthors** table:

```
SELECT au_fname, au_lname
FROM authors
WHERE au_id NOT IN (SELECT au_id FROM titleauthor);
```

NULL comparisons (IS NULL and IS NOT NULL)

For example, to see titles that have NULL year-to-date sales:

```
SELECT title_id, SUBSTRING(title, 1, 25) AS title
FROM titles
WHERE ytd_sales IS NULL;
```



Be sure *not* to specify = *NULL* in a query. *NULL* is unknown and can never be equal to anything. Using = *NULL* is not the same as specifying the *IS NULL* operator. Refer to the section [IS Operator](#), earlier in this chapter, for more information.

Pattern matches (LIKE and NOT LIKE)

For example, to see authors whose last names start with a "C":

```
SELECT au_id
FROM authors
WHERE au_lname LIKE 'C%';
```

Range operations (BETWEEN and NOT BETWEEN)

For example, to see authors with last names between "Smith" and "White":

```
SELECT au_lname, au_fname
FROM authors
WHERE au_lname BETWEEN 'smith' AND 'white';
```

Programming Tips and Gotchas

The *WHERE* clause may require special handling when dealing with certain datatypes, such as *LOBs* or certain character

sets like UNICODE.

Parentheses are used to control evaluation hierarchy within a *WHERE* clause. Encapsulating a clause within parentheses tells the DBMS to evaluate that clause before others. Parentheses can be nested to create a hierarchy of evaluations. The innermost parenthetical clause will be evaluated first. You should watch parentheses very carefully for two reasons:

- You must always have an equal number of opening and closing parentheses. Any imbalance in the number of opening and closing parentheses will cause an error.
- You should be careful where you place parentheses, since misplacing a parenthesis can dramatically change the result set of your query.

For example, consider again the following query, which returns six rows in the **pubs** database on the SQL Server platform:

```
SELECT DISTINCT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE ta.title_id IN (SELECT title_id FROM sales
                      WHERE qty >= 75)
OR (a.au_id IN (SELECT au_id FROM titleauthor
               WHERE royaltyper >= 60)
AND a.au_id IN (SELECT au_id FROM titleauthor
                WHERE au_ord = 2))
```

The output from this query is as follows:

```
au_id
-----
213-46-8915
724-80-9391
899-46-2035
998-72-3567
```

Changing just one set of parenthesis produces different results:

```
SELECT DISTINCT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE (ta.title_id IN (SELECT title_id FROM sales
                      WHERE qty >= 75)
OR a.au_id IN (SELECT au_id FROM titleauthor
               WHERE royaltyper >= 60))
AND a.au_id IN (SELECT au_id FROM titleauthor
                WHERE au_ord = 2)
```

This time, the output will look like this:

au_id

213-46-8915

724-80-9391

899-46-2035

DB2

Supported as described above.

MySQL

Supported as described above.

Oracle

Supported as described above.

PostgreSQL

Supported as described above.

SQL Server

Supported as described above.

See Also

ALL/ANY/SOME
BETWEEN
DECLARE CURSOR
DELETE
EXISTS
IN
LIKE
SELECT
UPDATE



Chapter 4. SQL Functions

A function is a special type of command word in the SQL command set, and each SQL dialect varies in its implementation of the command set. In effect, functions are one-word commands that return single values. The value of a function can be determined by input parameters, as with a function that averages a list of database values. But many functions do not use any type of input parameter, such as the function that returns the current system time, *CURRENT_TIME*.

The ANSI standard supports a number of useful functions. This chapter covers those functions, providing detailed descriptions and examples for each platform. In addition, each database maintains a long list of their own internal functions that are outside of the scope of the standard SQL. This chapter provides parameters and descriptions for each database implementation's internal functions.

In addition, most database platforms support the ability to create user-defined functions (UDFs). For more information on UDFs, refer to the [CREATE/ALTER FUNCTION/PROCEDURE Statements](#).

4.1 Types of Functions

There are different ways to categorize functions into groups. The following subsections describe distinctions that are critical to understand how functions work.

4.1.1 Deterministic and Nondeterministic Functions

Functions can be either *deterministic* or *nondeterministic*. A *deterministic* function always returns the same results if given the same input values. A *nondeterministic* function may return different results every time it is called, even when the same input values are provided.

Why is it important that a given input always returns the same output? It is important because of how functions may be used within views, in user-defined functions, and in stored procedures. Restrictions vary across implementations, but these objects sometimes allow only deterministic functions within their defining code. For example, SQL Server allows the creation of an index on a column expression—as long as the expression does not contain nondeterministic functions. Rules and restrictions vary between the platforms, so check the specific documentation when using functions.

4.1.2 Aggregate and Scalar Functions

Another way of categorizing functions is in terms of whether they operate on values from just one row at a time, on values from a collection, or on a set of rows. *Aggregate functions* operate against a collection of values and return a single summarizing value. *Scalar functions* return a single value based on scalar input arguments. Some scalar functions, such as *CURRENT_TIME*, do not require any arguments.

4.1.3 Window Functions

Window functions can be thought of as being similar to aggregate functions in that they operate over many rows at one time. The difference lies in how you define those rows. Aggregate functions operate over the sets of rows defined by a query's *GROUP BY* clause. With window functions, you specify the set of rows for each function call, and different invocations of a function within the same query can execute over different sets of rows.

4.2 ANSI SQL Aggregate Functions

Aggregate functions return a single value based upon a set of other values. If used among other expressions in the item list of a *SELECT* statement, the *SELECT* must have a *GROUP BY* or *HAVING* clause. No *GROUP BY* or *HAVING* clause is required if the aggregate function is the only value retrieved by the *SELECT* statement. The supported aggregate functions and their syntax are listed in [Table 4-1](#).

Table 4-1. ANSI SQL aggregate functions

Function	Usage
AVG(<i>expression</i>)	Computes the average value of a column given by <i>expression</i> .
CORR(<i>dependent, independent</i>)	Computes a correlation coefficient.
COUNT(<i>expression</i>)	Counts the rows defined by the <i>expression</i> .
COUNT(*)	Counts all rows in the specified table or view.
COVAR_POP (<i>dependent, independent</i>)	Computes population covariance.
COVAR_SAMP(<i>dependent, independent</i>)	Computes sample covariance.
CUME_DIST(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Computes the relative rank of a hypothetical row within a group of rows, where the rank is equal to the number of rows less than or equal to the hypothetical row divided by the number of rows in the group.
DENSE_RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a dense rank (no ranks are skipped) for a hypothetical row (<i>value_list</i>) in a group of rows generated by GROUP BY.

MIN(<i>expression</i>)	Finds the minimum value in a column given by <i>expression</i> .
MAX(<i>expression</i>)	Finds the maximum value in a column given by <i>expression</i> .
PERCENT_RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.
PERCENTILE_CONT (<i>percentile</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates an interpolated value that, if added to the group, would correspond to the <i>percentile</i> given.
PERCENTILE_DISC (<i>percentile</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Returns the value with the smallest cumulative distribution value greater than or equal to <i>percentile</i> .
RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a rank for a hypothetical row (<i>value_list</i>) in a group of rows generated by GROUP BY.
REGR_AVGX(<i>dependent, independent</i>)	Computes the average of the independent variable.
REGR_AVGY(<i>dependent, independent</i>)	Computes the average of the dependent variable.
REGR_COUNT(<i>dependent, independent</i>)	Counts the number of pairs remaining in the group after any pair with one or more NULL values has been eliminated.
REGR_INTERCEPT(<i>dependent, independent</i>)	Computes the y-intercept of the least-squares-fit linear equation.

REGR_R2(<i>dependent, independent</i>)	Squares the correlation coefficient.
REGR_SLOPE(<i>dependent, independent</i>)	Determines the slope of the least-squares-fit linear equation.
REGR_SXX(<i>dependent, independent</i>)	Sums the squares of the independent variables.
REGR_SXY(<i>dependent, independent</i>)	Sums the products of each pair of variables.
REGR_SYY(<i>dependent, independent</i>)	Sums the squares of the dependent variables.
STDDEV_POP(<i>expression</i>)	Computes the population standard deviation of all <i>expression</i> values in a group.
STDDEV_SAMP(<i>expression</i>)	Computes the sample standard deviation of all <i>expression</i> values in a group.
SUM(<i>expression</i>)	Computes the sum of column values given by <i>expression</i> .
VAR_POP(<i>expression</i>)	Computes the population variance of all <i>expression</i> values in a group.
VAR_SAMP(<i>expression</i>)	Computes the sample standard deviation of all <i>expression</i> values in a

expression)

group.

Technically speaking, *ANY*, *EVERY*, and *SOME* are considered aggregate functions. However, they have been discussed as range search criteria since they are most often used that way. Refer to [ALL/ANY/SOME Operators](#) for more information on these functions.

The number of values processed by an aggregate function varies depending on the number of rows queried from the table. This behavior makes aggregate functions different from scalar functions, which can only operate on the values of a single row per invocation.

The general syntax of an aggregate function is:

aggregate_function_name ([*ALL* | *DISTINCT*] *expression*)

The aggregate function name may be *AVG*, *COUNT*, *MAX*, *MIN*, or *SUM*, as listed in [Table 4-1](#). The *ALL* keyword, which specifies the default behavior, evaluates all rows when aggregating the value of the function. The *DISTINCT* keyword uses only distinct values when evaluating the function.



All aggregate functions except *COUNT*(*) will ignore NULL values when computing their results.

AVG and SUM

The *AVG* function computes the average of values in a column or an expression. *SUM* computes the sum. Both functions work with numeric values and ignore NULL values. Use the *DISTINCT* keyword to compute the average or sum of all distinct values of a column or expression.

SQL Standard Syntax

AVG ([*ALL* | *DISTINCT*] *expression*)

SUM ([*ALL* | *DISTINCT*] *expression*)

MySQL, PostgreSQL, and SQL Server

All these platforms support the SQL2003 syntax of *AVG* and *SUM*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and the following analytic syntax:

AVG ([*ALL* | *DISTINCT*] *expression*) *OVER* (*window_clause*)

SUM ([*ALL* | *DISTINCT*] *expression*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the [Section 4.3](#) later in this chapter.

Examples

The following query computes average year-to-date sales for each type of book:

```
SELECT type, AVG( ytd_sales ) AS "average_ytd_sales"FROM titles GROUP BY type;
```

This query returns the sum of year-to-date sales for each type of book:

```
SELECT type, SUM( ytd_sales ) FROM titles GROUP BY type;
```

CORR

The *CORR* function returns the correlation coefficient between a set of dependent and independent variables.

SQL2003 Syntax

Calls the function with two variables, one dependent and the other independent:

```
CORR( dependent, independent )
```

Any pair in which either the dependent variable, independent variable, or both are NULL is ignored. The result of the function is NULL when none of the input pairs consist of two non-NULL values.

Oracle

Oracle supports the SQL2003 syntax, and the following analytic syntax:

```
CORR ( dependent, independent ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the [Section 4.3](#) later in this chapter.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *CORR* function.

Example

The following *CORR* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;
```

Y	X
1	3
2	2
3	1

```
SELECT CORR(y,x) FROM test2;
```

```
CORR(Y,X)
```

```
-----  
-1
```

COUNT

The *COUNT* function is used to compute the number of rows in an expression.

SQL2003 Syntax

COUNT(*)

COUNT([ALL|DISTINCT] *expression*)

COUNT(*)

Counts all the rows in the target table whether or not they include NULLs.

COUNT([ALL|DISTINCT] *expression*)

Computes the number of rows with non-NULL values in a specific column or expression. When the keyword *DISTINCT* is used, duplicate values will be ignored and a count of the distinct values is returned. *ALL* returns the number of non-NULL values in the expression and is implicit when *DISTINCT* is not used.

MySQL, PostgreSQL, and SQL Server

All of these platforms support the SQL2003 syntax of *COUNT*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and the following analytic syntax:

COUNT ({*|[DISTINCT] *expression*}) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

Examples

This query counts all rows in a table:

```
SELECT COUNT(*) FROM publishers;
```

The following query finds the number of different countries where publishers are located:

```
SELECT COUNT(DISTINCT country) "Count of Countries"  
FROM publishers
```

COVAR_POP

The *COVAR_POP* function returns the population covariance of a set of dependent and independent variables.

SQL2003 Syntax

Call the function with two variables, one dependent and the other independent:

COVAR_POP(*dependent*, *independent*)

The function disregards any pair in which either the dependent variable, independent variable, or both are NULL. If no rows remain in the group after NULL elimination, then the result of the function is NULL.

Oracle

Oracle supports the SQL2003 syntax and implements the following analytic syntax:

COVAR_POP (*dependent*, *independent*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

DB2

In DB2, the function is named *CORRELATION*.

MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *COVAR_POP* function.

Example

The following *COVAR_POP* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;
```

```
      Y      X
-----
      1      3
      2      2
      3      1
```

```
SELECT COVAR_POP(y,x) FROM test2;
```

```
COVAR_POP(Y,X)
-----
      -.66666667
```

COVAR_SAMP

The *COVAR_SAMP* function returns the sample covariance of a set of dependent and independent variables.

SQL2003 Syntax

Call the function with two variables, one dependent and the other independent:

COVAR_SAMP(*dependent*, *independent*)

The function disregards any pair in which either the dependent variable, independent variable, or both are NULL. The result of the function is NULL when none of the input pairs consist of two non-NULL values.

Oracle

Oracle supports the SQL2003 syntax and implements the following analytic syntax:

COVAR_SAMP (*dependent*, *independent*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *COVAR_SAMP* function.

Example

The following *COVAR_SAMP* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;
```

```
   Y   X
-----
   1   3
   2   2
   3   1
```

```
SQL> SELECT COVAR_SAMP(y,x) FROM test2;
```

```
COVAR_SAMP(Y,X)
-----
             -1
```

CUME_DIST

Computes the relative rank of a hypothetical row within a group of rows, where that relative rank is computed as follows:

$$(rows_preceding_hypothetical + rows_peered_with_hypothetical) / rows_in_group$$

Bear in mind that the *rows_in_group* value includes the hypothetical row that you are proposing when you call the function.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [,expression...]
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
CUME_DIST OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *CUME_DIST* aggregate function.

Example

The following example determines the relative rank of the hypothetical new row (*num=4, odd=1*) within each group of rows from **test4**, where groups are distinguished by the values in the *odd* column:

SELECT * FROM test4;

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

SELECT odd, CUME_DIST(4,1) WITHIN GROUP (ORDER BY num, odd)

FROM test4

GROUP BY odd;

ODD	CUME_DIST(4,1) WITHIN GROUP (ORDER BY NUM, ODD)
0	1
1	.8

In group *odd=0*, the new row comes after the three rows: (0,0), (2,0), and (4,0). It will peer with itself. The total number of rows in the group will be four, which includes the hypothetical row. The relative rank, therefore, is computed as follows:

$$\begin{aligned} & (3 \text{ rows preceding} + 1 \text{ peering}) / (3 \text{ in group} + 1 \text{ hypothetical}) \\ & = 4 / 4 = 1 \end{aligned}$$

In group *odd=1*, the new row follows the three rows (1,1), (3,1), and a duplicate (3,1). Again, there is one peer, the hypothetical row itself. The number of rows in the group is five, which includes the hypothetical row. The relative rank is then:

$$\begin{aligned} & (3 \text{ rows preceding} + 1 \text{ peering}) / (4 \text{ in group} + 1 \text{ hypothetical}) \\ & = 4 / 5 = .8 \end{aligned}$$

DENSE_RANK

Computes a rank in a group for a hypothetical row that you supply. This is a *dense rank*. Rankings are never skipped, even when a group contains rows that rank identically.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [,expression...]
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
DENSE_RANK( ) OVER (partitioning ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *DENSE_RANK* aggregate function. However, DB2 does support *DENSE_RANK* as an analytic function. See the section later in this chapter titled [Section 4.3](#).

Example

The following example determines the dense rank of the hypothetical new row (*num=4, odd=1*) within each group of rows from **test4**, where groups are distinguished by the values in the *odd* column:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, DENSE_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
```

```
FROM test4
```

```
GROUP BY odd;
```

```
ODD DENSE_RANK(4,1) WITHIN GROUP (ORDER BY NUM, ODD)
```

0	4
1	3

In group `odd=0`, the new row comes after (0,0), (2,0), and (4,0), and thus it is position 4. In group `odd=1`, the new row follows (1,1), (3,1), and a duplicate (3,1). In that case, the duplicate occurrences of (3,1) both rank #2, so the new row is ranked #3. Compare this behavior with `RANK`, which gives a different result.

MIN and MAX

`MIN(expression)` and `MAX(expression)` find the minimum and maximum value of `expression` (string, datetime, or numeric) in a set of rows. `DISTINCT` or `ALL` may be used with these functions, but do not affect the result.

SQL2003 Syntax

`MIN([ALL | DISTINCT] expression)`

`MAX([ALL | DISTINCT] expression)`

PostgreSQL and SQL Server

These platforms support the SQL2003 syntax of `MIN` and `MAX`.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and implements the following analytic syntax:

`MIN ({ALL|[DISTINCT] expression}) OVER (window_clause)`

`MAX ({ALL|[DISTINCT] expression}) OVER (window_clause)`

For an explanation of the `window_clause`, see the section later in this chapter titled [Section 4.3](#).

MySQL

MySQL supports the SQL2003 syntax of `MIN` and `MAX`. MySQL also supports the functions `LEAST()` and `GREATEST()`, providing the same capabilities.

Examples

The following query finds the best and worst sales for any title on record:

```
SELECT MIN(ytd_sales), MAX(ytd_sales)
FROM titles;
```

Aggregate functions are used often in the `HAVING` clause of queries with `GROUP BY`. The following query selects all categories (types) of books that have an average price for all books in the category higher than \$15.00:

```
SELECT type 'Category', AVG( price ) 'Average Price'
FROM titles
GROUP BY type
HAVING AVG(price) > 15
```

PERCENT_RANK

Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [,expression...]
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following syntax:

```
PERCENT_RANK( ) OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *PERCENT_RANK* aggregate function.

Example

The following example determines the percentage rank of the hypothetical new row (*num*=4, *odd*=1) within each group of rows from **test4**, where groups are distinguished by the values in the *odd* column:

```
SELECT * FROM test4;
```

```
NUM    ODD
-----
0      0
1      1
2      0
3      1
3      1
4      0
5      1
```

```
SELECT odd, PERCENT_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
```

```
FROM test4
```

```
GROUP BY odd;
```

```
ODD PERCENT_RANK(4,1) WITHIN GROUP (ORDER BY NUM, ODD)
```

```
-----
0          1
1          .75
```

In group `odd=0`, the new row comes after (0,0), (2,0), and (4,0), and thus it is position 4. The rank computation is: $(4^{\text{th}} \text{rank} - 1) / 3 \text{ rows} = 100\%$. In group `odd=1`, the new row follows (1,1), (3,1), and a duplicate (3,1), and is again ranked at #4. The rank computation for `odd=1` is: $(4^{\text{th}} \text{rank} - 1) / 4 \text{ rows} = 3/4 = 75\%$.

PERCENTILE_CONT

Generates an interpolated value corresponding to a percentile that you specify.

SQL2003 Syntax

In the following syntax, *percentile* is a number between zero and one:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_list)
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_CONT (percentile) WITHIN GROUP
```

```
(ORDER BY sort_list) OVER (partitioning)
```

See [Section 4.3](#) later in this chapter for a description of partitioning.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_CONT*.

Example

The following example groups the data in `test4` by the column named `odd`, and invokes *PERCENTILE_CONT* to return a 50th percentile value for each group:

```
SELECT * FROM test4;
```

```
NUM  ODD
-----
0    0
1    1
2    0
3    1
3    1
4    0
5    1
```

```
SELECT odd, PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
FROM test4
GROUP BY odd;
      ODD PERCENTILE_CONT(0.50)WITHINGROUP(ORDERBYNUM)
```

```
-----
      0                2
      1                3
```

PERCENTILE_DISC

Determines the value in a group with the smallest cumulative distribution greater than or equal to a percentile that you specify.

SQL2003 Syntax

In the following syntax, *percentile* is a number between zero and one:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_list)
sort_list ::= sort_item [,sort_item...]
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_DISC (percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See [Section 4.3](#) later in this chapter for a description of partitioning.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_DISC*.

Example

The following example is similar to that for *PERCENTILE_CONT*, except that it returns, for each group, the value closest, but not exceeding, the 60th percentile:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, PERCENTILE_DISC(0.60) WITHIN GROUP (ORDER BY NUM)
```

```
FROM test4
```

```
GROUP BY odd;
```

```
PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
```

2
3

RANK

Computes a rank in a group for a hypothetical row that you supply. This is not a dense rank. If the group contains rows that rank identically, then it's possible for ranks to be skipped. If you want a dense rank, use the *DENSE_RANK* function.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [, expression...]
```

```
sort_list ::= sort_item [, sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
RANK( ) OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *RANK* aggregate function.

Example

The following example determines the rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from `test4`, where groups are distinguished by the values in the `odd` column:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
```

```
FROM test4
```

```
GROUP BY odd;
```

ODD	RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
0	4
1	4

In both cases, the rank of the hypothetical new row is 4. In group `odd=0`, the new row comes after: `(0,0)`, `(2,0)`, and `(4,0)`, and thus it is position 4. In group `odd=1`, the new row follows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`. In that case, the new row is preceding by three rows, so it is ranked #4. Compare this behavior with `DENSE_RANK`.

The REGR Family of Functions

SQL2003 defines a family of functions, having names beginning with `REGR_`, that relate to different aspects of linear regression. The functions work in the context of a least-squares regression line.

SQL2003 Syntax

Following is the syntax and a brief description of each `REGR_` function:

```
REGR_AVGX( dependent, independent )
```

Averages (as in `AVG(x)`) the *independent* variable values.

```
REGR_AVGY( dependent, independent )
```

Averages (as in `AVG(y)`) the *dependent* variable values.

```
REGR_COUNT( dependent, independent )
```

Counts the number of non-NULL number pairs.

`REGR_INTERCEPT(dependent, independent)`

Computes the y-intercept of the regression line.

`REGR_R2(dependent, independent)`

Computes the coefficient of determination.

`REGR_SLOPE(dependent, independent)`

Computes the slope of the regression line.

`REGR_SXX(dependent, independent)`

Sums the squares of the independent variable values.

`REGR_SXY(dependent, independent)`

Sums the products of each pair of values.

`REGR_SYY(dependent, independent)`

Sums the squares of the dependent variable values.

The `REGR_` functions only work on number pairs containing two non-NULL values. Any number pair with one or more NULL values will be ignored.

DB2 and Oracle

DB2 and Oracle support the SQL2003 syntax for all `REGR_` functions. In addition, DB2 allows the shortened name `REGR_ICPT` in place of `REGR_INTERCEPT`.

Oracle supports the following analytic syntax:

`REGR_function (dependent, independent) OVER (window_clause)`

For an explanation of the `window_clause`, see the section later in this chapter titled [Section 4.3](#).

MySQL, PostgreSQL, and SQL Server

These platforms do not implement the `REGR` family of functions.

Example

The following `REGEXP_COUNT` example demonstrates that any pair with one or more NULL values is ignored. The table `test3` contains three non-NULL number pairs, and three other pairs having at least one NULL:

SQL> SELECT * FROM test3;

Y	X
1	3
2	2
3	1
4	NULL
NULL	4
NULL	NULL

The *REGR_COUNT* function ignores the pairs having NULLs, counting only those pairs with non-NULL values:

```
SELECT REGR_COUNT(y,x) FROM test3;
```

```
REGR_COUNT(Y,X)
```

```
-----
```

```
3
```

Likewise, all other *REGR_* functions filter out any pairs having NULL values before performing their respective computations.

STDDEV_POP

Use *STDDEV_POP* to find the *population standard deviation* within a group of numeric values.

SQL2003 Syntax

```
STDDEV_POP( numeric_expression )
```

DB2 and MySQL

Use the *STDDEV* function. In DB2 and MySQL, *STDDEV* returns the population standard deviation.

PostgreSQL

This platform does not provide a function to compute population standard deviation.

Oracle

Oracle supports the standard syntax and the following analytic syntax:

```
STDDEV_POP( numeric_expression ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

SQL Server

Use the *STDEV* function.

Example

The following example computes the population standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;
```



```
X
-----
1
2
3
```

```
SELECT STDDEV_POP(x) FROM test;
```

```
STDDEV_POP(X)
```

```
-----
.816496581
```

STDDEV_SAMP

Use *STDDEV_SAMP* to find the *sample standard deviation* within a group of numeric values.

SQL2003 Syntax

```
STDDEV_SAMP( numeric_expression )
```

Oracle

Oracle supports the standard syntax. Oracle also provides the *STDDEV* function, which operates similar to *STDDEV_SAMP* except that it returns zero as not NULL when there is only one value in the set.

Oracle also supports analytic syntax:

```
STDDEV_SAMP( numeric_expression ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

DB2

This platform does not provide a function to compute sample standard deviation.

MySQL

MySQL does not provide a function to compute sample standard deviation. MySQL does provide a function named *STDDEV*, but it returns the *population* standard deviation.

PostgreSQL

Use *STDDEV*.

SQL Server

Use *STDEV* (with only one *D!*).

Example

The following example computes the sample standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;
```

```
X
-----
1
2
3

SELECT STDDEV_SAMP(x) FROM test;

STDDEV_SAMP(X)
-----
1
```

VAR_POP

Use *VAR_POP* to compute the population variance of a set of values.

SQL2003 Syntax

VAR_POP(*numeric_expression*)

DB2 and PostgreSQL

These platforms do not provide a function to compute population variance.

MySQL

Use the *VARIANCE* function, which in MySQL returns the population variance.

Oracle

Oracle supports the standard syntax and the following analytic syntax:

VAR_POP (*numeric_expression*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

SQL Server

Use the *VARP* function.

Example

The following example computes the population variance for the values 1, 2, and 3:

```
SELECT * FROM test;
```

```
X
-----
1
2
3
```

```
SELECT VAR_POP(x) FROM test;
```

```
VAR_POP(X)
```

```
-----
.66666667
```

VAR_SAMP

Use *VAR_SAMP* to compute the sample variance of a set of values.

SQL2003 Syntax

```
VAR_SAMP( numeric_expression )
```

DB2 and PostgreSQL

Use *VARIANCE(numeric_expression)* to compute sample variance.

MySQL

MySQL provides no function for computing sample variance. There is the *VARIANCE* function, but in MySQL that function returns the *population* variance.

Oracle

Oracle supports the standard syntax. You may also use the *VARIANCE* function, which differs from *VAR_SAMP* by returning zero (and not NULL) for sets having only a single value.

Oracle also supports analytic syntax:

```
VAR_SAMP( numeric_expression ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

SQL Server

Use the *VAR* function.

Example

The following example computes the sample variance for the values 1, 2, and 3:

```
SELECT * FROM test;
```

X

1

2

3

SELECT VAR_SAMP(x) FROM test;

VAR_SAMP(X)

1



< Day Day Up >



4.3 ANSI SQL Window Functions

SQL2003 allows for a *window_clause* in aggregate function calls, the addition of which makes those functions into window functions. Both Oracle and DB2 support this window function syntax. This section describes how to use the *window_clause* within Oracle and DB2.



Oracle tends to refer to window functions as *analytic functions*.

Window, or analytic, functions are similar to standard aggregate functions in that they operate on multiple rows, or groups of rows, within the result set returned from a query. However, the groups of rows that a window function operates on are defined not by a GROUP BY clause, but by partitioning and windowing clauses. Furthermore, the order within these groups is defined by an ordering clause, but that order only affects function evaluation, and has no effect on the order in which rows are returned by the query.



Window functions are the last items in a query to be evaluated except for the *ORDER BY* clause. Because of this late evaluation, window functions *cannot* be used within the *WHERE*, *GROUP BY*, or *HAVING* clauses.

4.3.1 SQL2003's Window Syntax

SQL2003 specifies the following syntax for window functions:

FUNCTION_NAME(*expr*) OVER {*window_name*}(*window_specification*)}

window_specification ::= [*window_name*][*partitioning*][*ordering*][*framing*]

partitioning ::= PARTITION BY *value* [, *value*...] [COLLATE *collation_name*]

ordering ::= ORDER [SIBLINGS] BY *rule* [, *rule*...]

rule ::= {*value*|*position*|*alias*} [ASC|DESC] [NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {*start*|*between*} [*exclusion*]

start ::= {UNBOUNDED PRECEDING|*unsigned-integer* PRECEDING|CURRENT ROW}

between ::= BETWEEN *bound* AND *bound*

bound ::= {*start*|UNBOUNDED FOLLOWING|*unsigned-integer* FOLLOWING}

exclusion ::= {EXCLUDE CURRENT ROW|EXCLUDE GROUP
|EXCLUDE TIES|EXCLUDE NO OTHERS}

4.3.2 Oracle's Window Syntax

Oracle's window function syntax is as follows:

FUNCTION_NAME(*expr*) OVER (*window_clause*)

window_clause ::= [*partitioning*] [*ordering* [*framing*]]

partitioning ::= PARTITION BY *value* [, *value*...]

ordering ::= ORDER [SIBLINGS] BY *rule* [, *rule*...]

rule ::= {*value*|*position*|*alias*} [ASC|DESC]
[NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {*not_range*|*begin* AND *end*}

not_range ::= {UNBOUNDED PRECEDING
|CURRENT ROW
|*value* PRECEDING}

begin ::= {UNBOUNDED PRECEDING
|CURRENT ROW
|*value* {PRECEDING|FOLLOWING}}

end ::= {UNBOUNDED FOLLOWING
|CURRENT ROW
|*value* {PRECEDING|FOLLOWING}}

4.3.3 DB2's Window Syntax

DB2's syntax is similar to Oracle's. For OLAP, ranking, and numbering functions, DB2 allows the following syntax:

FUNCTION_NAME(*expr*) OVER (*window_clause*)

window_clause ::= [*partitioning*] [*ordering*]

partitioning ::= PARTITION BY (*value* [, *value*...])

ordering ::= {ORDER BY *rule* [, *rule*...] | ORDER OF *table_name*}

rule ::= {*value*|*position*|*alias*} [ASC|DESC [NULLS {FIRST|LAST}]]

When aggregate functions (e.g. AVG) are used as window functions, DB2 allows the addition of a *framing* clause:

FUNCTION_NAME(*expr*) OVER (*window_clause*)

window_clause ::= [*partitioning*] [*ordering* [*framing*]] [*all*/*framing*]

all ::= RANGE UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

partitioning ::= PARTITION BY (*value* [, *value*...])

ordering ::= {ORDER BY *rule* [, *rule*...] | ORDER OF *table_name*}

rule ::= {*value*|*position*|*alias*} [ASC|DESC [NULLS {FIRST|LAST}]]

framing ::= {ROWS|RANGE} {*group_start*|*group_between*|*group_end*}

group_start ::= {UNBOUNDED PRECEDING|*unsigned-integer* PRECEDING
|CURRENT ROW}

group_between ::= BETWEEN {UNBOUNDED PRECEDING|*unsigned_integer* PRECEDING
|*unsigned_integer* FOLLOWING|CURRENT ROW}
AND {UNBOUNDED FOLLOWING|*unsigned_integer* PRECEDING
|*unsigned_integer* FOLLOWING|CURRENT ROW}

group_end ::= UNBOUNDED FOLLOWING|*unsigned-integer* FOLLOWING}

4.3.4 Partitioning

Partitioning the rows operated on by the partition clause is similar to using the *GROUP BY* expression on a standard *SELECT* statement. The partitioning clause takes a list of expressions that will be used to divide the result set into groups. We'll use the following table as the basis for some examples:

SELECT * FROM odd_nums;

NUM	ODD
0	0
1	1
2	0
3	1

The following results illustrate the effects of partitioning by **ODD**. The sum of the even numbers is 2 (0+2), and the sum of the odd numbers is 4 (1+3). The second column of the result set reports the sum of all values in the partition to which *that row* belongs. Yet all the detail rows are returned. The query provides summary results in the context of detail rows:

```
SELECT NUM, SUM(NUM) OVER (PARTITION BY ODD) S  
FROM ODD_NUMS;
```

NUM	S
0	2
2	2
1	4
3	4

Not using a partitioning clause at all will sum all of the numbers in the **NUM** column for each row returned by the query. In effect, the entire result set is treated as a single, large partition:

```
SELECT NUM, SUM(NUM) OVER ( ) S FROM ODD_NUMS;
```

NUM	S
0	6
1	6
2	6
3	6

4.3.5 Ordering

You specify the order of the rows on which an analytic function operates using the *ordering* clause. However, this analytic ordering clause does not define the result set ordering. To define the overall result set ordering, you must use the query's *ORDER BY* clause. The following use of Oracle's *FIRST_VALUE* function illustrates the effect of different orderings of the partitions:

```
SELECT NUM,  
    SUM(NUM) OVER (PARTITION BY ODD) S,  
    FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM ASC) first_asc,
```



```
FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM DESC) first_desc  
FROM ODD_NUMS;
```

NUM	S	FIRST_ASC	FIRST_DESC
0	2	0	2
2	2	0	2
1	4	1	3
3	4	1	3

As you can see, the *ORDER BY* clauses in the window function invocations affect the ordering of the rows in the respective partitions when those functions are evaluated. Thus, *ORDER BY NUM ASC* orders partitions in ascending order, resulting in **0** for the first value in the even-number partition and **1** for the first value in the odd-number partition. *ORDER BY NUM DESC* has the opposite effect.



The preceding query also illustrates an important point: using window functions, you can summarize and order many different ways in the same query.

4.3.6 Grouping or Windowing

Many analytic functions also allow you to specify a virtual, moving window surrounding a row within a partition. You do this using the *framing* clause. Such moving windows are useful for running calculations such as a running total.

The following, Oracle-based example uses the *framing* clause on the analytic variant of *SUM* to calculate a running sum of the values in the first column. No partitioning clause is used, so each invocation of *SUM* operates over the entire result set. However, the *ORDER BY* clause sorts the rows for *SUM* in ascending order of **NUM**'s value, and the *BETWEEN* clause (which is the windowing clause) causes each invocation of *SUM* to include values for **NUM** only up through the current row. Each successive invocation of *SUM* includes yet another value for **NUM**, in order, from the lowest value of **NUM** to the greatest:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS  
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS;
```

NUM	S
0	0
1	1
2	3
3	6

This example's a bit too easy, as the order of the final result set happens to match the order of the running total. That doesn't need to be the case. The following example generates the same results, but in a different order. You can see that the running total values are appropriate for each value of **NUM**, but the rows are presented in a different order than before. The result set ordering is completely independent of the ordering used for window function calculations:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS  
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS  
ORDER BY NUM DESC;
```

NUM	S
3	6
2	3
1	1
0	0

4.3.7 List of Window Functions

SQL2003 specifies that any aggregate function may also be used as a window function. Both Oracle and DB2 largely follow the standard in that respect, so you'll find that you can take just about any aggregate function (certainly the standard ones) and apply to it the window function syntax described in the preceding sections.

In addition to the aggregate functions, SQL2003 defines the window functions described in the following sections. Only Oracle and DB2 currently implement these functions. All examples use the following table and data, which is a variation on the **ODD_NUMS** table used earlier to illustrate the concepts of partitioning, ordering, and grouping:

SELECT * FROM test4;

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

Platform-specific window functions for Oracle (there are none for DB2) are included in the lists found under [Section 4.5](#) later in this chapter.

CUME_DIST()

Calculates the cumulative distribution, or relative rank, of the current row to other rows in the same partition. The calculation for a given row is as follows:

$\text{number of peer or preceding rows} / \text{number of rows in partition}$

Because the result for a given row depends on the number of rows preceding that row in the same partition, it's important to always specify an *ORDER BY* clause when invoking this function.

SQL2003 Syntax

CUME_DIST() OVER {*window_name*}{*window_specification*}

DB2

DB2 does not support the *CUME_DIST()* window function.

Oracle

Oracle does not allow the framing portion of the windowing syntax. Oracle requires the *ordering* clause:

CUME_DIST() OVER (*[partitioning]* *ordering*)

Example

The following Oracle-based example uses *CUME_DIST()* to generate a relative rank for each row, ordering by **NUM**, after partitioning the data by **ODD**:

```
SELECT NUM, ODD, CUME_DIST( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
```

```
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	.333333333
2	0	.666666667
4	0	1
1	1	.25
3	1	.75
3	1	.75
5	1	1

Following is an explanation of the calculation behind the rank for the row in which **NUM=0**:

1. Because of the *ORDER BY* clause, the rows in the partition are ordered as follows:

```
NUM=0
NUM=2
NUM=4
```

1. There are no rows preceding **NUM=0**.
2. There is one row that is a peer of **NUM=0**, and that is the **NUM=0** row itself. Thus, the divisor is 1.
3. There are three rows in the partition as a whole, making the dividend 3.
4. The result of 1/3 is .33 repeating, as shown in the example output.

DENSE_RANK()

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question, and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same. Unlike the case with *RANK()*, gaps in rank numbers will not result from two rows sharing the same rank.

SQL2003 Syntax

DENSE_RANK() OVER {*window_name*({*window_specification*)}

DB2

DB2 requires the *ordering* clause and does not allow the *framing* clause:

DENSE_RANK() OVER ([*partitioning*] *ordering*)

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

DENSE_RANK() OVER ([*partitioning*] *ordering*)

Example

Compare the results from the following Oracle-based example to those shown in the section on the *RANK()* function:

SELECT NUM, DENSE_RANK() OVER (ORDER BY NUM) rank

FROM test4;

NUM	RANK
0	1
1	2
2	3
3	4
3	4
4	5
5	6

The two rows where NUM=3 are both ranked at #3. The next higher row is ranked at #4. Rank numbers are not skipped, hence the term "dense."

RANK()

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question, and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same, and will lead to subsequent gaps in rank numbers.

SQL2003 Syntax

RANK() OVER {*window_name*({*window_specification*)}

DB2

DB2 requires the *ordering* clause and does not allow the *framing* clause:

`RANK() OVER ([partitioning] ordering)`

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

`RANK() OVER ([partitioning] ordering)`

Example

The following Oracle-based example uses the **NUM** column to rank the rows in the **test4** table:

SELECT NUM, RANK() OVER (ORDER BY NUM) rank

FROM test4;

NUM	RANK
0	1
1	2
2	3
3	4
3	4
4	6
5	7

Because both rows where **NUM**=3 rank the same at #4, the next higher row will be ranked at #6. The #5 rank is skipped.

PERCENT_RANK

Computes the relative rank of a row by dividing that row's rank less 1 by the number of rows in the partition, also less 1:

$(rank - 1) / (rows - 1)$

Compare this calculation to that used for *CUME_DIST*.

SQL2003 Syntax

`PERCENT_RANK() OVER ({window_name}(window_specification))`

DB2

DB2 does not support the *PERCENT_RANK()* window function.

Oracle

Oracle requires the *ordering* clause and does not allow the *framing* clause:

`PERCENT_RANK() OVER ([partitioning] ordering)`

Example

The following, Oracle-based example assigns a relative rank to values of **NUM**, partitioning the data on the **ODD** column:

```
SELECT NUM, ODD, PERCENT_RANK( ) OVER
      (PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	0
2	0	.5
4	0	1
1	1	0
3	1	.333333333
3	1	.333333333
5	1	1

Following is an explanation of the calculation behind the rank for the row in which **NUM=2**:

1. Row **NUM=2** is the second row in its partition; thus, it ranks #2.
2. Subtract 1 from 2 to get a divisor of 1.
3. The dividend is the total number of rows in the partition, or 3.
4. Subtract 1 from 3 to get a dividend of 2.
5. The result of 1/3 is .33 repeating, as shown in the example.

ROW_NUMBER

Assigns a unique number to each row in a partition.

SQL2003 Syntax

`ROW_NUMBER() OVER ({window_name}(window_specification))`

DB2

DB2 does not allow the *framing* clause, and it makes the *ordering* clause optional:

`ROW_NUMBER() OVER ([partitioning] [ordering])`

Oracle

Oracle requires the *ordering* clause and does not allow the *framing* clause:

`ROW_NUMBER() OVER ([partitioning] ordering)`

Example

```
SELECT NUM, ODD, ROW_NUMBER( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	1
2	0	2
4	0	3
1	1	1
3	1	2
3	1	3
5	1	4



4.4 ANSI SQL Scalar Functions

The ANSI SQL scalar functions return a single value each time they are invoked. The SQL standard provides many scalar functions that can be used to manipulate date and time types, strings, and numbers, as well as retrieve system information such as the current user or login name. Scalar functions fall into the categories listed in [Table 4-2](#).

Table 4-2. Categories of scalar functions

Function category	Explanation
Built-in	Performs operations on values or settings built into the database. Oracle uses the term "built-in" to describe all the specialty functions that are provided by Oracle, and thus "built into" their DBMS. This is a distinct and separate usage from the "built-in" functions described here.
CASE and CAST	While these two functions operate on scalar input values, they are in a category all their own. <i>CASE</i> supplies <i>IF-THEN</i> logic to SQL statements and <i>CAST</i> can convert values from one datatype to another.
Date and Time	Performs operations on <i>temporal</i> datatypes and returns values in a <i>temporal</i> datatype format. There is no SQL2003 function that operates on a temporal datatype and returns a temporal result. The closest function is <i>EXTRACT</i> (covered in Section 4.4.3 later in this chapter), which operates on temporal values and returns numeric values. Functions returning temporal values but operating on no arguments are covered in the later Section 4.4.1 .
Numeric	Performs operations on numeric values and returns numeric values.
String	Performs operations on character values (e.g., <i>CHAR</i> , <i>VARCHAR</i> , <i>NCHAR</i> , <i>NVARCHAR</i> , and <i>CLOB</i>) and returns a string or numeric value.

4.4.1 Built-in Scalar Functions

SQL2003 built-in scalar functions identify both the current user session and the characteristics of the current user session, such as the current session privileges. Built-in scalar functions are always nondeterministic. The *CURRENT_DATE*, *CURRENT_TIME*, and *CURRENT_TIMESTAMP* functions listed in [Table 4-3](#) are built-in functions that fall into the date-and-time category of functions. Although the five platforms provide many additional functions beyond these SQL built-ins, the SQL standard defines only those listed in [Table 4-3](#).

Table 4-3. ANSI SQL built-in scalar functions

Function	Usage
<i>CURRENT_DATE</i>	Returns the current date.
<i>CURRENT_TIME</i>	Returns the current time.
<i>CURRENT_TIMESTAMP</i>	Returns the current date and time.
<i>CURRENT_USER</i> or <i>USER</i>	Returns the currently active user within the database server.
<i>SESSION_USER</i>	Returns the currently active Authorization ID, if it differs from the user.
<i>SYSTEM_USER</i>	Returns the currently active user within the host operating system.

DB2

DB2 supports the SQL2003 built-in scalar functions *CURRENT_DATE*, *CURRENT_TIME*, *CURRENT_TIMESTAMP*, *CURRENT_USER*, and *USER*. The SQL2003 scalar functions *CURRENT_ROLE*, *LOCAL_TIME*, *LOCAL_TIMESTAMP*, *SESSION_USER*, and *SYSTEM_USER* are not supported.

MySQL

MySQL supports all the SQL2003 built-in scalar functions, plus Oracle's variant *SYSDATE*. In addition, MySQL supports *NOW()* as a synonym of the function *CURRENT_TIMESTAMP*.

Oracle

Oracle supports *USER* and *SYSDATE* as a synonym of *CURRENT_TIMESTAMP*.

PostgreSQL

PostgreSQL supports all the SQL2003 built-in scalar functions except *SESSION_USER*.

SQL Server

SQL Server supports all the built-in scalar functions.

Examples

The following queries retrieve the values from built-in functions. Notice that the various platforms return dates in their native formats:

```
/* On MySQL */
```

```
SELECT CURRENT_TIMESTAMP;'2001-12-15 23:50:26'
```

```
/* DB2 */
```

```
VALUES CURRENT_TIMESTAMP'2001-12-15 23.50.26.000000'
```

```
/* On Microsoft SQL Server */
```

```
SELECT CURRENT_TIMESTAMPGO'Dec 15,2001 23:50:26'
```

```
/* On Oracle */
```

```
SELECT USER FROM dual;dylan
```

4.4.2 CASE and CAST Functions

ANSI SQL2003 provides a function named *CASE* that can be used to create *IF-THEN* flow-control logic within queries or update statements. The *CAST* function is for converting between datatypes and is also in the ANSI standard. All of the databases covered by this book provide ANSI standard support for both the *CASE* and *CAST* functions.

CASE

The *CASE* function provides *IF-THEN-ELSE* functionality within a *SELECT* or *UPDATE* statement. It evaluates a list of conditions and returns one value out of several possible values.

CASE has two usages: simple and searched. Simple *CASE* expressions compares one value, the *input_value*, with a list of other values and return a result associated with the first matching value. Searched *CASE* expressions allow the analysis of several logical conditions and return a result associated with the first one that is true.

All vendors provide the ANSI SQL2003 syntax for *CASE*.

SQL2003 Syntax and Description

```
-- Simple comparison operation
```

```
CASE input_value WHEN when_condition THEN resulting_value[...n]
```

```
[ELSE else_result_value]
```

```
END

-- Boolean searched operation

CASE

WHEN Boolean_condition THEN resulting_value[...n]

[ELSE else_result_expression]

END
```

In the simple *CASE* function, the *input_value* is evaluated against each *WHEN* clause. The *resulting_value* is returned for the first TRUE instance of *input_value = when_condition*. If no *when_condition* evaluates as TRUE, the *else_result_value* is returned. If no *else_result_value* is specified, then NULL is returned.

In the more elaborate Boolean searched operation, the structure is essentially the same as the simple comparison operation except that each *WHEN* clause has its own Boolean comparison operation.

In either usage, multiple *WHEN* clauses are used, though only one *ELSE* clause is necessary.

Examples

Here is a simple comparison operation where the *CASE* function alters the display of the **contract** column to make it more understandable:

```
SELECT au_fname,
       au_lname,
       CASE contract
         WHEN 1 THEN 'Yes'
         ELSE 'No'
       END 'contract'
FROM   authors
WHERE  state = 'CA'
```

Here is an elaborate searched *CASE* function in a *SELECT* statement that will report how many titles have been sold in different year-to-date sales ranges:

```
SELECT CASE
         WHEN ytd_sales IS NULL THEN 'Unknown'
         WHEN ytd_sales <= 200 THEN 'Not more than 200'
         WHEN ytd_sales <= 1000 THEN 'Between 201 and 1000'
         WHEN ytd_sales <= 5000 THEN 'Between 1001 and 5000'
         WHEN ytd_sales <= 10000 THEN 'Between 5001 and 10000'
         ELSE 'Over 10000'
       END 'YTD Sales',
       COUNT(*) 'Number of Titles'
FROM   titles
GROUP BY CASE
         WHEN ytd_sales IS NULL THEN 'Unknown'
         WHEN ytd_sales <= 200 THEN 'Not more than 200'
         WHEN ytd_sales <= 1000 THEN 'Between 201 and 1000'
```

```
        WHEN ytd_sales <= 5000 THEN 'Between 1001 and 5000'
        WHEN ytd_sales <= 10000 THEN 'Between 5001 and 10000'
        ELSE 'Over 10000'
    END
ORDER BY MIN( ytd_sales )
```

The results are:

YTD Sales	Number of Titles

Unknown	2
Not more than 200	1
Between 201 and 1000	2
Between 1001 and 5000	9
Between 5001 and 10000	1
Over 10000	3

Next is an *UPDATE* statement that applies discounts to all of the titles. This more complicated command will discount all personal computer-related titles by 25%, all other titles by 10%, and apply only a 5% discount to titles with year-to-date sales exceeding 10,000 units. This query uses a searched *CASE* expression to perform price adjustment:

```
UPDATE titles
SET price = price *
    CASE
        WHEN ytd_sales > 10000 THEN 0.95 -- 5% discount
        WHEN type = 'popular_comp' THEN 0.75 -- 25% discount
        ELSE 0.9 -- 10% discount
    END
WHERE pub_date IS NOT NULL
```

The update has now completed three separate *UPDATE* operations in a single statement.

CAST

The *CAST* command explicitly converts an expression of one datatype to another. All vendors provide the ANSI SQL2003 syntax for *CAST*.

SQL2003 Syntax and Description

```
CAST(expression AS data_type[(length)])
```

The *CAST* function converts any expression, such as a column value or variable, into another defined datatype. The length of the datatype may be optionally supplied for those datatypes (such as *CHAR* or *VARCHAR*) that support lengths.



Be aware that some conversions, such as *DECIMAL* values to *INTEGER*, will result in rounding operations. Also, some conversion operations may result in an error if the new datatype does not have sufficient space to display the converted value.

Examples

This example retrieves the year-to-date sales as a *CHAR* and concatenates it with a literal string and a portion of the title of the book. It converts **ytd_sales** to *CHAR(5)*, plus it shortens the length of the **title** to make the results more readable.

```
SELECT CAST(ytd_sales AS CHAR(5)) + ' Copies sold of ' + CAST(title AS VARCHAR(30))
FROM titles
WHERE ytd_sales IS NOT NULL
AND ytd_sales > 10000
ORDER BY ytd_sales DESC
```

The results are:

```
-----
22246 Copies sold of The Gourmet Microwave
18722 Copies sold of You Can Combat Computer Stress
15096 Copies sold of Fifty Years in Buckingham Pala
```

4.4.3 Numeric Scalar Functions

The list of official SQL2003 numeric functions is rather small, and the different platforms provide supplementary mathematical and statistical functions. MySQL directly supports many of these functions. The other database platforms offer the same capabilities of numeric scalar functions through their own internally defined functions, but they do not share the same name as those declared by the SQL standard. The supported numeric functions and syntax are listed in [Table 4-4](#).

Table 4-4. SQL2003 numeric functions

Function	Usage
<i>ABS</i>	Returns the absolute value of a number.
<i>MOD</i>	Returns the remainder of one number divided into another.
<i>BIT_LENGTH</i>	Returns an integer value representing the number of bits in another value.
<i>CEIL</i> or <i>CEILING</i>	Rounds a noninteger value upwards to the next greatest integer. Returns an integer value unchanged.
<i>CHAR_LENGTH</i>	Returns an integer value representing the number of characters in a string expression.
<i>EXP</i>	Raises a value to the power of the mathematical constant known as <i>e</i> .
<i>EXTRACT</i>	Allows the datepart to be extracted (<i>YEAR</i> , <i>MONTH</i> , <i>DAY</i> , <i>HOUR</i> , <i>MINUTE</i> , <i>SECOND</i> , <i>TIMEZONE_HOUR</i> , or <i>TIMEZONE_MINUTE</i>) from a temporal expression.
<i>FLOOR</i>	Rounds a noninteger value downwards to the next least integer. Returns an integer value unchanged.
<i>LN</i>	Returns the natural logarithm of a number.
<i>OCTET_LENGTH</i>	Returns an integer value representing the number of octets in another value. This value is the same as <i>BIT_LENGTH/8</i> .

<i>POSITION</i>	Returns an integer value representing the starting position of a string within the search string.
<i>POWER</i>	Raises a number to a specified power.
<i>SQRT</i>	Computes the square root of a number.
<i>WIDTH_BUCKET</i>	Deposits a value into the appropriate bucket from a set of buckets covering a given range.

ABS

All platforms have standard SQL2003 support for the *ABS* function.

SQL2003 Syntax

ABS(*expression*)

ABS returns the absolute value of the number in *expression*.

Example

The following shows how to use the *ABS* function:

```
/* SQL2003 */
```

```
SELECT ABS(-1) FROM NUMBERS1
```

BIT_LENGTH, CHAR_LENGTH, and OCTET_LENGTH

All platforms stray from the ANSI standard in their support for the scalar functions for determining the length of expressions. While the platform support is nonstandard, the equivalent functionality exists under different names.

SQL2003 Syntax

The SQL2003 scalar functions for getting the length of a value take an *expression* to calculate the value and return the length as an integer. The *BIT_LENGTH* function returns the number of bits contained within the value of *expression*. The *CHAR_LENGTH* is the number of characters in the string *expression*. *OCTET_LENGTH* returns the number of octets within the string *expression*. All three of these functions will return NULL if *expression* is NULL.

BIT_LENGTH(*expression*)

CHAR_LENGTH(*expression*)

OCTET_LENGTH(*expression*)

DB2

DB2 does not support *BIT_LENGTH*, *CHAR_LENGTH*, or *OCTET_LENGTH*. DB2 does support a scalar function named *LENGTH*(), which provides similar functionality for nongraphic string types. For graphic string types, *LENGTH* returns the number of double-byte characters in a string.

MySQL

MySQL supports *CHAR_LENGTH* and the SQL2003 synonym *CHARACTER_LENGTH*().

Oracle

The closest any of the platforms get to the *BIT_LENGTH* function is Oracle. Oracle supports the *LENGTHB* function, which returns an integer value representing the number of bytes in an expression. For the length of an expression in characters, Oracle provides a *LENGTH()* function as a synonym for *CHAR_LENGTH*.

PostgreSQL

PostgreSQL supports *CHAR_LENGTH* and the SQL2003 synonym *CHARACTER_LENGTH()*.

SQL Server

SQL Server provides the *LEN* function.

Example

The following example, shown for different databases, determines the length of a string and a value retrieved from a column:

```
/* On MySQL and PostgreSQL */
```

```
SELECT CHAR_LENGTH('hello');SELECT OCTET_LENGTH(book_title) FROM titles;
```

```
/* On Microsoft SQL Server */
```

```
SELECT DATALENGTH(title) FROM titlesWHERE type = 'popular_comp'GO
```

```
/* On Oracle or DB2 */
```

```
SELECT LENGTH('HORATIO') "Length of characters"FROM dual;
```

CEIL

The *CEIL* function returns the smallest integer greater than an input value that you specify.

SQL2003 Syntax

SQL2003 supports the following two forms of the function:

```
CEIL( expression )
```

```
CEILING ( expression )
```

DB2

DB2 supports both variations of the SQL2003 syntax.

MySQL

MySQL supports only *CEILING*.

Oracle

Oracle supports only *CEIL*.

PostgreSQL

PostgreSQL has no support for either *CEIL* or *CEILING*.

SQL Server

SQL Server supports only *CEILING*.

Examples

When you pass a positive, noninteger number, the effect of *CEIL* is to round up to the next highest integer:

```
SELECT CEIL(100.1) FROM dual;
```

```
FLOOR(100.1)
```

```
-----  
      101
```

Remember, though, that with negative numbers, rounding "up" results in a lower absolute value:

```
SELECT CEIL(-100.1) FROM dual;
```

```
FLOOR(-100.1)
```

```
-----  
     -100
```

Use *FLOOR* to get behavior opposite to that of *CEIL*.

EXP

The *EXP* function returns the value of the mathematical constant *e* (approximately 2.718281) raised to the power of a specified number.

SQL2003 Syntax

All platforms support the SQL2003 syntax:

```
EXP( expression )
```

Example

The following example uses *EXP* to return an approximation of *e*:

```
SELECT EXP(1) FROM dual;
```

```
EXP(1)  
-----  
2.71828183
```

Use *LN* to go in the opposite direction.

EXTRACT

The *EXTRACT* function is supported by Oracle, PostgreSQL, and MySQL. The other platforms support a separate command to accomplish the same functionality.

SQL2003 Syntax

EXTRACT(date_part FROM expression)

The SQL2003 scalar function for extracting parts from a date is *EXTRACT*. The SQL2003 *EXTRACT* function takes a *date_part* and an *expression* that evaluates to a date time value.

DB2

DB2 provides unique functions that provide the same behavior as *EXTRACT*. Those functions are *DAY*, *DAYNAME*, *DAYOFWEEK*, *DAYOFWEEK_ISO*, *DAYOFYEAR*, *DAYS*, *HOURL*, *JULIAN_DAY*, *MICROSECOND*, *MIDNIGHT_SECONDS*, *MINUTE*, *MONTH*, *MONTHNAME*, *SECOND*, *TIME*, *WEEK*, *WEEK_ISO*, and *YEAR*. Look to the list shown later in this chapter in [Section 4.5.1](#) for examples on how to use these functions. As with Oracle, DB2 also offers a *TO_CHAR* function that can be used to format a string from a date value.



When extracting parts of a date value on DB2 into an integer, such as the year or minute, use the special functions provided by DB2 instead of the *TO_CHAR* function. Doing so will provide the highest performance, since a conversion from a result string into an integer won't be required.

MySQL

MySQL's implementation is extended somewhat beyond the ANSI standard. The ANSI standard does not have a provision for returning multiple fields from the same call to *EXTRACT()* (e.g., *DAY_HOUR*). The MySQL extensions try to accomplish what the combination *DATE_TRUNC()* and *DATE_PART()* do in PostgreSQL. MySQL supports the dateparts listed in [Table 4-5](#).

Table 4-5. MySQL dateparts

Type value	Meaning
<i>SECOND</i>	Seconds
<i>MINUTE</i>	Minutes
<i>HOURL</i>	Hours
<i>DAY</i>	Days
<i>MONTH</i>	Months
<i>YEAR</i>	Years
<i>MINUTE_SECOND</i>	Minutes and seconds
<i>HOURL_MINUTE</i>	Hours and minutes
<i>DAY_HOUR</i>	Days and hours
<i>YEAR_MONTH</i>	Years and months
<i>HOURL_SECOND</i>	Hours, minutes, and seconds
<i>DAY_MINUTE</i>	Days, hours, and minutes
<i>DAY_SECOND</i>	Days, hours, minutes, and seconds

Oracle

Oracle supports the SQL2003 syntax with the dateparts listed in [Table 4-6](#).

Table 4-6. Oracle dateparts

Type value	Meaning
<i>DAY</i>	The day of the month field (1-31)

<i>HOUR</i>	Returns the hour field (0-23)
<i>MINUTE</i>	The minutes field (0-59)
<i>MONTH</i>	The month field (1-12)
<i>SECOND</i>	The seconds field (0-59)
<i>TIMEZONE_HOUR</i>	The hour component of the time zone offset
<i>TIMEZONE_MINUTE</i>	The minute component of the time zone offset
<i>TIMEZONE_REGION</i>	The current time zone name
<i>TIMEZONE_ABBR</i>	The abbreviation of the current time zone
<i>YEAR</i>	The year field

PostgreSQL

PostgreSQL supports the SQL2003 syntax with a few extra dateparts. PostgreSQL supports the dateparts listed in [Table 4-7](#).

Table 4-7. PostgreSQL dateparts

Type value	Meaning
<i>CENTURY</i>	The year field divided by 100.
<i>DAY</i>	The day of the month field (1-31).
<i>DECADE</i>	The year field divided by 10.
<i>DOW</i>	Day of week (0-6, where Sunday is 0). This type only works for <i>TIMESTAMP</i> values.
<i>DOY</i>	Day of year (1-366.) The maximum returned value is only 365 for years that are not leap years. This type can only be used on <i>TIMESTAMP</i> values.
<i>EPOCH</i>	Returns the number of seconds between the epoch (1970-01-01 00:00:00-00) and the value. The result can be negative for values before the epoch.
<i>HOUR</i>	Returns the hour field (0-23).
<i>MICROSECONDS</i>	The seconds field (including fractional parts) multiplied by 1,000,000.
<i>MILLENNIUM</i>	The year field divided by 1,000.
<i>MILLISECONDS</i>	The seconds field (including fractional parts) multiplied by 1,000.
<i>MINUTE</i>	The minutes field (0-59).
<i>MONTH</i>	The month field (1-12).
<i>QUARTER</i>	The quarter of the year (1-4) that the value is in. This type can only be used with <i>TIMESTAMP</i> values.
<i>SECOND</i>	The seconds field (0-59).
<i>TIMEZONE_HOUR</i>	The hour component of the time zone offset.
<i>TIMEZONE_MINUTE</i>	The minute component of the time zone offset.
<i>WEEK</i>	The week number that the value falls on within the year.
<i>YEAR</i>	The year field.

SQL Server

SQL Server provides the function *DATEPART*(*date_part*, *expression*) as a synonym for the SQL2003 function *EXTRACT*(*date_part* FROM *expression*). SQL Server supports the dateparts listed in [Table 4-8](#).

Table 4-8. SQL Server dateparts

Type value	Meaning
------------	---------

<i>year</i>	The year field of the datetime expression. The abbreviations <i>yy</i> and <i>yyyy</i> can also be used for two-digit and four-digit years, respectively.
<i>quarter</i>	The quarter that the datetime expression falls on. The abbreviations <i>q</i> and <i>qq</i> can also be used.
<i>dayofyear</i>	The day of the year for the date time expression. The abbreviations <i>y</i> and <i>dy</i> can also be used.
<i>day</i>	The day of the month for the date time expression. The abbreviations <i>d</i> and <i>dd</i> can also be used.
<i>week</i>	The week of the year for the date time expression. The abbreviations <i>wk</i> and <i>ww</i> can also be used.
<i>weekday</i>	The day of the week for the date time expression. The abbreviation <i>dw</i> can also be used.
<i>hour</i>	The hour of the day for the date time expression. The abbreviation <i>hh</i> can also be used.
<i>minute</i>	The minute of the hour for the date time expression. The abbreviations <i>n</i> and <i>mi</i> can also be used.
<i>second</i>	The second of the minute for the date time expression. The abbreviations <i>s</i> and <i>ss</i> can also be used.
<i>millisecond</i>	The millisecond for the date time expression. The abbreviation <i>ms</i> can also be used.

Example

This example extracts dateparts from several date time values:

```
/* On MySQL */
```

```
SELECT EXTRACT(YEAR FROM "2013-07-02");2013
```

```
SELECT EXTRACT(YEAR_MONTH FROM "2013-07-02 01:02:03");201307
```

```
SELECT EXTRACT(DAY_MINUTE FROM "2013-07-02 01:02:03");20102
```

```
/* On PostgreSQL */
```

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
```

20

FLOOR

The *FLOOR* function returns the largest integer less than an input value that you specify.

SQL2003 Syntax

All platforms support the SQL2003 syntax:

```
FLOOR( expression )
```

Examples

When you pass a positive number, the effect of *FLOOR* is to eliminate anything after the decimal point:

```
SELECT FLOOR(100.1) FROM dual;
```

```
FLOOR(100.1)
```

```
-----
```

```
100
```

Remember, though, that with negative numbers, going in the "less than" direction corresponds to increasingly larger absolute values:

```
SELECT FLOOR(-100.1) FROM dual;
```

```
FLOOR(-100.1)
```

```
-----
```

```
-101
```

Use *CEIL* to get behavior opposite to *FLOOR*.

LN

The *LN* function returns the natural logarithm of a number, which is the power to which you would need to raise the mathematical constant *e* (approximately 2.718281) in order to get the number in question as the result.

SQL2003 Syntax

```
LN( expression )
```

DB2, Oracle, PostgreSQL

DB2, Oracle, and PostgreSQL support the SQL2003 syntax for the *LN* function. DB2 and PostgreSQL also support the use of *LOG* as a synonym for *LN*.

MySQL and SQL Server

MySQL and SQL Server call their natural logarithm function *LOG*:

```
LOG( expression )
```

Example

The following, Oracle-based example shows the natural logarithm of a number closely approximating the mathematical constant known as *e*:

```
SELECT LN(2.718281) FROM dual;
```

```
LN(2.718281)
```

```
-----
```

```
.999999695
```

Use the *EXP* function to go in the other direction.

MOD

The *MOD* function returns the remainder of a dividend divided by a divider. All platforms support the SQL2003 syntax for the *MOD* function.

SQL2003 Syntax

```
MOD( dividend, divider )
```

The standard syntax for the *MOD* function is to return the remainder of dividend divided by the divider; returns the dividend if the divider is 0.

Example

The following shows how to use the *MOD* function from within a *SELECT* statement:

```
SELECT MOD(12, 5) FROM NUMBERS
```

```
2
```

POSITION

The *POSITION* function returns an integer that indicates the starting position of a string within the search string.

SQL2003 Syntax

```
POSITION( string1 IN string2 )
```

The standard syntax for the *POSITION* function is to return the first location of *string1* within *string2*. *POSITION* returns 0 if *string1* does not occur within *string2* and NULL if either argument is NULL.

DB2

DB2's equivalent function is *POSSTR*.

MySQL

MySQL supports *POSITION* as defined in SQL2003.

Oracle

Oracle's equivalent function is called *INSTR*.

PostgreSQL

PostgreSQL supports *POSITION* as defined in SQL2003.

SQL Server

SQL Server has both *CHARINDEX* and *PATINDEX*. *CHARINDEX* and *PATINDEX* are very similar, except that *PATINDEX* allows the use of wildcard characters in the search criteria.

Examples

```
/* DB2 */
```

```
SELECT POSSTR('bar', 'foobar');
```

```
4
```

```
/* On MySQL */
```

```
SELECT LOCATE('bar', 'foobar');4
```

/ On MySQL and PostgreSQL */*

```
SELECT POSITION('fu' IN 'snafu');0
```

/ On Microsoft SQL Server */*

```
SELECT CHARINDEX( 'de', 'abcdefg' )GO4
```

```
SELECT PATINDEX( '%fg', 'abcdefg' )GO6
```

POWER

Use *POWER* to raise a number to a specific value.

SQL2003 Syntax

```
POWER( base, exponent )
```

The result of the *POWER* function is *base* raised to the *exponent* power, or *base^{exponent}*. If *base* is negative, then *exponent* must be an integer.

DB2, Oracle, PostgreSQL, and SQL Server

These vendors all support the SQL2003 syntax.

MySql

MySQL supports the same functionality, but requires the use of the keyword *POW*:

```
POW( base, exponent )
```

Example

Raising a positive number to an exponent is straightforward:

```
SELECT POWER(10,3) FROM dual;
```

```
POWER(10,3)
```

```
-----
```

```
1000
```

Anything raised to the 0th power evaluates to 1:

```
SELECT POWER(0,0) FROM dual;
```

```
POWER(0,0)
```

```
-----
```

```
1
```

Negative exponents move the decimal point to the left:

```
SELECT POWER(10,-3) FROM dual;
```

```
POWER(10,-3)
```

```
-----
```

```
.001
```

SQRT

The *SQRT* function returns the square root of a number.

SQL2003 Syntax

All platforms support the SQL2003 syntax:

```
SQRT( expression )
```

Example

```
SELECT SQRT(100) FROM dual;
```

```
SQRT(100)
```

```
-----
```

```
10
```

WIDTH_BUCKET

The *WIDTH_BUCKET* function assigns values to buckets in an equiwidth histogram.

SQL2003 Syntax

In the following syntax, *expression* represents a value to be assigned to a bucket. You would typically base *expression* on one or more columns returned by a query.

```
WIDTH_BUCKET(expression, min, max, buckets)
```

The *buckets* argument specifies the number of buckets to create over the range defined by *min* through *max*. *min* is inclusive, whereas *max* is not. The value from *expression* is assigned to one of those buckets, and the function then returns the corresponding bucket number. When *expression* falls outside the range of buckets, the function returns either 0 or *max*+1, depending on whether *expression* is lower than *min* or greater than or equal to *max*.

Examples

The following example divides the integer values 1 through 10 into two buckets:

```
SELECT x, WIDTH_BUCKET(x,1,10,2)
```

```
FROM pivot;
```

X WIDTH_BUCKET(X,1,10,2)

1	1
2	1
3	1
4	1
5	1
6	2
7	2
8	2
9	2
10	3

The following example is more interesting. It divides 11 values from 1 through 10 into three buckets, and illustrates the distinction between *min* being inclusive and *max* being noninclusive:

SELECT x, WIDTH_BUCKET(x,1,10,3)

FROM pivot;

X WIDTH_BUCKET(X,1,10,3)

1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	3
9.9	3
10	4

Pay particular attention to the results for X=1, 9.9, and 10. An input value of *min*, 1 in this example, falls into that first bucket, proving that the lower end of the range for bucket #1 is defined as $x \geq \text{min}$. An input value of *max*, however, falls *outside* the highest bucket. In this example, 10 falls into the overflow bucket numbered $\text{max}+1$. The value 9.9 falls into bucket #3, illustrating that the upper end of the range for the highest bucket is defined as $x < \text{max}$.

4.4.4 String Functions and Operators

Basic string functions and operators offer a number of capabilities and return a string value as a result. Some string functions are dyadic, indicating that they operate on two strings at once. SQL2003 supports the string functions listed in [Table 4-9](#).

Table 4-9. SQL String functions and operators

Function or operator	Usage
Concatenation operator	Appends two or more literal string expressions, column values, or variables together into one string.
<i>CONVERT</i>	Converts a string to a different representation within the same character set.
<i>LOWER</i>	Converts a string to all lowercase characters.
<i>OVERLAY</i>	Returns the result of replacing a substring of one string with another.
<i>SUBSTRING</i>	Returns a portion of a string.
<i>TRANSLATE</i>	Converts a string from one character set to another.
<i>TRIM</i>	Removes leading characters, trailing characters, or both from a character string.
<i>UPPER</i>	Converts a string to all uppercase characters.

Concatenation Operator

SQL2003 defines a concatenation operator (||), which joins two distinct strings into one string value.

DB2

DB2 supports the SQL2003 concatenation operator and the *CONCAT* function as a synonym.

MySQL

MySQL supports *CONCAT*() as a synonym for the SQL2003 concatenation operator.

Oracle and PostgreSQL

PostgreSQL and Oracle support the SQL2003 double vertical bar concatenation operator.

SQL Server

SQL Server uses the plus sign (+) as a synonym for the SQL2003 concatenation operator. SQL Server has the system setting *CONCAT_NULL_YIELDS_NULL*, which can be set to alter the behavior when NULL values are used in the concatenation of string values.

Examples

```
/* SQL2003 Syntax */
```

```
'string1' || 'string2' || 'string3'
```

```
'string1string2string3'
```

```
/* On MySQL */
```

```
CONCAT('string1', 'string2')
```

```
'string1string2'
```

If any of the concatenation values are NULL, the entire returned string is NULL. Also, if a numeric value is concatenated, it is implicitly converted to a character string.

```
SELECT CONCAT('My ', 'bologna ', 'has ', 'a ', 'first ', 'name...');
```

```
'My bologna has a first name...'
```

```
SELECT CONCAT('My ', NULL, 'has ', 'first ', 'name...');NULL
```


CONVERT and TRANSLATE

The *CONVERT* function alters the representation of a character string within its character set and collation. For example, *CONVERT* might be used to alter the number of bits per character.

TRANSLATE alters the character set of a string value from one base-character set to another. Thus, *TRANSLATE* might be used to translate a value from the English character set to a Kanji (Japanese) or Cyrillic (Russian) character set. The translation must already exist, either by default or by having been created using the *CREATE TRANSLATION* command.

SQL2003 Syntax

CONVERT (*char_value* USING *conversion_char_name*)

TRANSLATE(*char_value* USING *translation_name*)

CONVERT converts *char_value* to the character set with the name supplied in *conversion_char_name*. *TRANSLATE* converts *char_value* to the character set provided in *translation_name*.

Among the platforms, only Oracle supports *CONVERT* and *TRANSLATE* with the same meaning as SQL2003. Oracle's implementation of *TRANSLATE* is very similar to SQL2003, but not identical. In its implementation, Oracle accepts only two arguments and translates only between the database character set and the national language support character set.

MySQL's implementation of the *CONV* function only translates numbers from one base to another. In contrast, SQL Server's implementation of *CONVERT* is a very rich utility that alters the base datatype of an expression, but is otherwise dissimilar to the SQL2003 *CONVERT* function. PostgreSQL does not support *CONVERT*, and its implementation of *TRANSLATE* morphs any occurrence of a character string to any other character string.

DB2

DB2 *does not* support *CONVERT*, and its support for the *TRANSLATE* function is not ANSI standard. *TRANSLATE* is used to translate substrings and has historically acted as a synonym for *UPPER*, since *UPPER* was only recently added to DB2. When *TRANSLATE* is used with only a string expression as an argument, the result will be an uppercase equivalent of that argument. When used with more than one argument, as in *TRANSLATE*(*source*, *replacements*, *match*), the function translates all characters in *source* that are also in *match*. Each character in *source* will be replaced with the character in *replacements* that is found in the same position as the matching character in *match*. Following are some examples:

TRANSLATE('Hello, World!')

'HELLO, WORLD!'

TRANSLATE('Hello, World!', 'wZ', 'IW')

'Hewwo, Zorwd!'

MySQL

MySQL *does not* support *TRANSLATE* or *CONVERT*.

Oracle

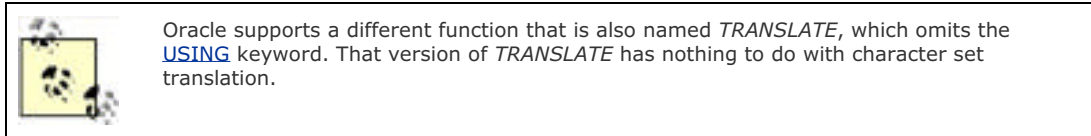
Oracle supports the following *CONVERT* and *TRANSLATE* functions:

CONVERT(*char_value*, *target_char_set*, *source_char_set*)

TRANSLATE(*char_value* USING {*CHAR_CS* | *NCHAR_CS*})

Under Oracle's implementation, the *CONVERT* function returns the text of *char_value* in the target character set. The *char_value* is the string to convert, *target_char_set* is the name of the character set into which the string is to be converted, and *source_char_set* is the name of the character set in which *char_value* was originally stored.

Oracle's *TRANSLATE* function follows the ANSI syntax, but you can choose between two character sets only: the database character set (*CHAR_CS*) and the national character set (*NCHAR_CS*).



Both the target and source character set names can be passed either as literal strings, in variables, or in columns from a table. Note that replacement characters might be substituted when converting from or to a character set that does not support a representation of all the characters used in the conversion.

Oracle supports several common character sets, including *US7ASCII*, *WE8DECDEC*, *WE8HP*, *F7DEC*, *WE8EBCDIC500*, *WE8PC850*, and *WE8ISO8859P1*. For example:

```
SELECT CONVERT('Gro2', 'US7ASCII', 'WE8HP')FROM DUAL;Gross
```

PostgreSQL

PostgreSQL supports the ANSI standard *CONVERT* and conversions can be defined by using *CREATE CONVERSION*. PostgreSQL's implementation of the *TRANSLATE* function offers a large superset of functions that can convert any occurrence of one text string to another within another specified string.

```
TRANSLATE (character_string, from_text, to_text)
```

Here are some examples:

```
SELECT TRANSLATE('12345abcde', '5a', 'XX');1234XXbcde'
```

```
SELECT TRANSLATE(title, 'Computer', 'PC')FROM titlesWHERE type = 'Personal_computer'
```

```
SELECT CONVERT('PostgreSQL' USING iso_8859_1_to_utf_8)
```

```
'PostgreSQL'
```

SQL Server

SQL Server *does not* support *TRANSLATE*. SQL Server's implementation of the *CONVERT* function does not follow the SQL2003 specification. Instead, it is functionally equivalent to the *CAST* function.

```
CONVERT (data_type[(length) | (precision,scale)], expression[,style])
```

The *style* clause is used to define the format of a date conversion. Refer to the SQL Server documentation for more information. Following is an example:

```
SELECT title, CONVERT(char(7), ytd_sales)
```

```
FROM titles
```

```
ORDER BY title
```

```
GO
```

LOWER and UPPER

The functions *LOWER* and *UPPER* allow the case of a string to be altered quickly and easily, so that all the characters are lower- or uppercase, respectively. These functions are supported in all the database implementations covered in this book. The different database platforms also support a variety of other text formatting functions that are specific to their implementation.

SQL2003 Syntax

LOWER(string)

UPPER(string)

LOWER converts *string* into a lowercase string. *UPPER* is the uppercase counterpart of *LOWER*.

DB2 and MySQL

These platforms support the SQL2003 *UPPER* and *LOWER* scalar functions, as well as the synonyms *UCASE* and *LCASE*.

Oracle, PostgreSQL, and SQL Server

These platforms support the SQL2003 *UPPER* and *LOWER* scalar functions, as in the following example:

Example

```
SELECT LOWER('You Talkin To ME?'), UPPER('you talking to me?!');
```

```
you talkin to me?, YOU TALKING TO ME?!
```

OVERLAY

The *OVERLAY* function embeds one string into another and returns the result.

SQL2003 Syntax

OVERLAY(string PLACING embedded_string FROM start[FOR length])

If any of the inputs are NULL, the *OVERLAY* function returns a NULL. The *embedded_string* replaces the *length* characters in *string* starting at character position *start*. If the *length* is not specified, then the *embedded_string* will replace all characters after *start* in *string*.

DB2, MySQL, Oracle, and SQL Server

These platforms do not have support for the *OVERLAY* function. You can simulate the *OVERLAY* function on these platforms by using a combination of *SUBSTRING* and the concatenation operator.

PostgreSQL

PostgreSQL supports the ANSI standard for *OVERLAY*.

Examples

This is an example of how to use the *OVERLAY* function:

```
/* SQL2003 and PostgreSQL */
```

```
SELECT OVERLAY('DONALD DUCK' PLACING 'TRUMP' FROM 8) FROM NAMES;'DONALD TRUMP'
```

SUBSTRING

The *SUBSTRING* function allows one character string to be returned from another.

SQL2003 Syntax

```
SUBSTRING(extraction_string FROM starting_position [FOR length]
```

```
[COLLATE collation_name])
```

If any of the inputs are NULL, the *SUBSTRING* function returns a NULL. The *extraction_string* is the source where the character value is to be extracted. It may be a literal string, a column in a table with a character datatype, or a variable with a character datatype. The *starting_position* is an integer value telling the function at which position to begin performing the extract. The optional *length* is an integer value that tells the function how many characters to extract, starting at the *starting_position*. If the optional *FOR* is omitted, then the substring starting at *starting_position* and continuing to the end of the *extraction_string* is returned.

DB2

```
SUBSTR(extraction_string, starting_position [, length])
```

DB2's implementation, *SUBSTR*, largely functions the same way as SQL2003's *SUBSTRING*. It does not support the *COLLATE* clause. If *length* is omitted, the remainder of the string (starting at *starting_position*) is returned.

MySQL

```
SUBSTRING(extraction_string FROM starting_position)
```

MySQL's implementation assumes that the characters are to be extracted from the starting position and will continue to the end of the character string.

Oracle

```
SUBSTR(extraction_string, starting_position [, length])
```

Oracle's implementation, *SUBSTR*, largely functions the same way as SQL2003's *SUBSTRING*. It *does not* support the *COLLATE* clause. When a *starting_position* is a negative number, Oracle counts from the end of the *extraction_string*. If *length* is omitted, the remainder of the string (starting at *starting_position*) is returned.

PostgreSQL

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

PostgreSQL largely supports the ANSI standard, except that it does not accept the *COLLATE* clause.

SQL Server

`SUBSTRING(extraction_string [FROM starting_position] [FOR length])`

SQL Server largely supports the ANSI standard, except that it does not allow the *COLLATE* clause. SQL Server allows this command to be applied to text, image, and binary datatypes; however, the *starting_position* and *length* represent the number of bytes rather than the number of characters to count.

Examples

These examples generally work on any one of the five database platforms profiled in this book. Only the second Oracle example, with a negative starting position, fails on the others (assuming, of course, that Oracle's *SUBSTR* is translated into *SUBSTRING*):

/ On Oracle, counting from the left */*

```
SELECT SUBSTR('ABCDEFGF',3,4) FROM DUAL;'CDEF'
```

/ On Oracle, counting from the right */*

```
SELECT SUBSTR('ABCDEFGF',-5,4) FROM DUAL;'CDEF'
```

/ On MySQL */*

```
SELECT SUBSTRING('Be vewy, vewy quiet' FROM 5);'wy, vewy quiet'
```

/ On PostgreSQL or SQL Server */*

```
SELECT au_iname, SUBSTRING(au_fname, 1, 1)FROM authorsWHERE au_iname = 'Carson'
```

Carson C

TRIM

The *TRIM* function removes leading characters, trailing characters, or both from a specified character string or *BLOB* value. This function also removes other types of characters from a specified character string. The default function is to trim the specified character from both sides of the character string. If no removal character is specified, *TRIM* removes spaces by default.

SQL2003 Syntax

```
TRIM( [ [ {LEADING | TRAILING | BOTH} ] [removal_char] FROM ]
```

```
target_string
```

```
[COLLATE collation_name])
```

The *removal_char* is the character to be stripped out. The *target_string* is the character string from which characters are to be stripped. If a *removal_char* is not specified, then *TRIM* strips out spaces. The *COLLATE* clause forces the result set of the function into another pre-existing collation set.

DB2

DB2 provides the functions *LTRIM* and *RTRIM* to trim off leading spaces or trailing spaces, respectively.

MySQL, Oracle, and PostgreSQL

These platforms support the SQL2003 syntax of *TRIM*.

Microsoft SQL Server

SQL Server provides the functions *LTRIM* and *RTRIM* to trim off leading spaces or trailing spaces, respectively. On SQL Server the *LTRIM* and *RTRIM* cannot be used to trim other types of characters.

Examples

```
SELECT TRIM(' wamalamadingdong ');wamalamadingdong'
```

```
SELECT LTRIM( RTRIM(' wamalamadingdong '));wamalamadingdong'
```

```
SELECT TRIM(LEADING '19' FROM '1976 AMC GREMLIN');76 AMC GREMLIN'
```

```
SELECT TRIM(BOTH 'x' FROM 'xxxWHISKEYxxx');'WHISKEY'
```

```
SELECT TRIM(TRAILING 'snack' FROM 'scooby snack');scooby '
```

4.5 Platform-Specific Extensions

The following sections provide a full listing and description of each vendor-supported function. The functions are platform-specific. Thus, a MySQL function, for example, is not guaranteed to be supported by any other vendor.

4.5.1 DB2-Supported Extensions

This section provides an alphabetical listing of DB2-supported functions that are specific to DB2 and not from the ANSI standard, with examples and corresponding results.

ABSVAL(number)

Synonym for *ABS(number)*. For example:

```
VALUES( ABSVAL( -1 ) ) -> 1
```

ACOS(number)

Returns the arc cosine of a number ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

ASCII(text)

Returns the ASCII code of the first character of text. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(number, nbr)

Returns the arctangent of *number* and *nbr*. The values for *number* and *nbr* are not restricted, but the results range from $-\pi$ to π and are expressed in radians. *ATAN2(x,y)* is similar to *ATAN(y/x)* with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
VALUES( ATAN2( 3.1415, 1 ) ) -> +3.08177595443792E-001
```

ATANH(number)

Returns the hyperbolic arctangent of *number*. The values for *number* are not restricted, but the results range

from - to and are expressed in radians. For example:

VALUES(ATANH(0)) -> +0.00000000000000E+000

BIGINT(expression)

Returns a 64-bit integer representation of *expression*, where *expression* can be a number, character string, date, time, or timestamp. For example:

SELECT BIGINT('1991-11-22') -> 19911122

BLOB(string [,length])

Returns a *BLOB* representation of *string*, where *string* can be a string representation of any type. This example will return *BLOBs* from the **employee_images** table that contain the binary string "JFIF."

SELECT image FROM employee_images WHERE image LIKE BLOB('%JFIF%')

CHAR(expression [,length])

Returns a fixed-length character string representation of *expression*, where the optional parameter *length* is the length attribute of the character value returned.

CHR(number)

Returns the character having the ASCII code value equal to *number*. For example:

VALUES(CHR(120)) -> 'x'

CLOB(string [,length])

Returns a *CLOB* representation of *string*.

COALESCE(expression [,...])

Returns the first argument that is not NULL. For example:

SELECT COALESCE(1,2,3) -> 1

CONCAT(string1, string2)

Returns *string1* concatenated with *string2*. It is equivalent to the concatenation operator (||). For example:

SELECT CONCAT(au_lname, au_fname) FROM authors -> 'JeffersonThomas'

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

COSH(number)

Returns the hyperbolic cosine of *number*. For example:

VALUES(COSH(3.1415)) -> +1.15908832931176E+001

COT(number)

Returns the cotangent of *number*. For example:

SELECT COT(3.1415) -> -10792.88993953

DATE(expression)

Returns a date value from *expression*. If *expression* is an integer value, then the returned date is equivalent to *expression-1* days after January 1, 0001. For example:

SELECT DATE(3) -> '01/03/0001'

DAY(expression)

Returns the day number in a date expression. For example:

SELECT DAY('1999-04-15') -> 15

DAYNAME(expression)

Returns the name of the day in *expression* using the locale of the database server. For example:

SELECT DAYNAME('1999-04-15') -> 'Thursday'

DAYOFWEEK(expression), DAYOFWEEK_ISO(expression), DAYOFYEAR(expression)

Returns the day of the week or year in *expression*. *DAYOFWEEK* returns an integer value in the range 1-7, where 1 represents Sunday. *DAYOFWEEK_ISO* has the same range as *DAYOFWEEK*, but the value of 1 represents Monday. *DAYOFYEAR* returns an integer value in the range 1-366, where the value returned is the day of the year starting with 1 for January 1st. For example:

VALUES(DAYOFWEEK('1999-04-15')) -> 5

VALUES(DAYOFWEEK_ISO('1999-04-15')) -> 4

VALUES(DAYOFYEAR('1999-04-15')) -> 105

DAYS(expression)

Returns one more than the number of days between *expression* and January 1, 0001. For example:

VALUES(DAYS('1999-04-15')) -> 729859

DBCLOB(expression [,length])

Returns a *DBCLOB* representation of a graphic string type, where *length* specifies the length of the *DBCLOB* value returned.

DBPARTITIONNUM(column)

Returns the database partition number of the row containing *column*. For more information, please look to the DB2 user documentation for *DBPARTITIONNUM*.

DECIMAL(expression [,precision [,scale [,decimal_char]]], DEC(expression [,precision [,scale [,decimal_char]])

Returns a decimal representation of *expression*. The optional arguments control the *precision* and *scale* used in the result, whereas *decimal_char* is the decimal character found in *expression*, if any.

DECRYPT_BIN(*data* [, *password*]) or *DECRYPT_CHAR*(*data* [, *password*])

Returns decrypted data using the optional *password*. The result of *DECRYPT_BIN* is a *VARCHAR FOR BIT DATA*, whereas *DECRYPT_CHAR* returns a *VARCHAR*.

DEGREES(*number*)

Returns the number of degrees converted from the argument expressed in radians. For example:

VALUES(DEGREES(3.1415926)) -> +1.79999996929531E+002

DEREF(*expression*)

Returns the object reference of *expression*, where *expression* must return a *REF* to an object. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

DIFFERENCE(*expression1*, *expression2*)

Returns the difference between the sounds of two strings based on their *SOUNDEX* values. The result is an integer in the range of 0-4, where 4 is the best *SOUNDEX* match. For example:

VALUES(DIFFERENCE('thimble', 'nimble')) -> 4

DIGITS(*number*)

Returns the *number* argument into a character string of digits. The result of *DIGITS* does not contain decimal or sign characters and it may be padded on the left with zeros. For example:

VALUES(DIGITS(DEC('-3.1415926', 12, 8))) -> 000031415926

DOUBLE(*number*)

Returns *number* converted to a *DOUBLE*.

ENCRYPT(*data* [, *password* [, *hint*]])

Returns *data* encrypted with the optional *password*. The optional *hint* argument allows a user to store up to a 32-byte password hint that is encoded into the result value. To get the password hint, look to the *GETHINT* function. For example:

UPDATE employee SET ssn = ENCRYPT(ssn, 'luvbug', 'Herbie is-a?')

WHERE empid = '54321-AD'

EVENT_MON_STATE(*expression*)

Returns the state of an event monitor. For details on usage or available event monitors, please look to the DB2 user documentation.

FLOAT(*number*)

FLOAT is a synonym for *DOUBLE*.

GETHINT(*data*)

Returns the password hint encoded in *data*. The *data* argument must be encrypted with the *ENCRYPT* function. For example:

SELECT GETHINT(ssn) FROM EMPLOYEE WHERE empid = '54321-AD'

'Herbie is-a?'

GENERATE_UNIQUE()

Returns a 13-byte *CHAR(13) FOR BIT DATA* value that is guaranteed to be unique from other invocations within the same database. For example:

```
INSERT INTO employee(id, emp_name)
VALUES(GENERATE_UNIQUE( ), 'Bob Smith')
```

GRAPHIC(expression [,length])

Returns a fixed-length graphic string representation of *expression*, where the optional *length* argument is the length of the result.

HASHEDVALUE(column)

Returns the partitioning map index of the row containing *column*. For more information, please look to the DB2 user documentation for *HASHEDVALUE()*.

HEX(expression)

Returns a hexadecimal representation of *expression*. For example:

```
VALUES( HEX( 255 ) ) -> 000000FF
```

HOUR(expression)

Returns the hour of part of the time value in *expression*. For example:

```
SELECT HOUR(execution_time), COUNT(*) FROM trades
GROUP BY HOUR(execution_time)
```

```
1      2
-----
      8      2058
      9      856
     10      912
     11      714
...

```

IDENTITY_VAL_LOCAL()

Returns the value most recently assigned to an identity column. Returns NULL when an identity column has not been inserted into a table since the last *COMMIT* or *ROLLBACK* statement.

INSERT(expression1, expression2, expression3, expression4)

Returns *expression1* with *expression4*, replacing *expression3* bytes at position *expression2*. For example:

```
VALUES (INSERT('food', 2, 2, 'are')) -> 'fared'
```

INTEGER(expression)

Returns an integer representation of *expression*.

JULIAN_DAY(expression)

Returns the Julian day number for the date value in *expression*. The value returned is the number of days between the date value in *expression* and the start of the Julian date calendar, January 1, 4713 B.C. For example:

VALUES(JULIAN_DAY('1999-04-15')) -> 2451284

LEFT(string, length)

Returns the first *length* bytes from *string*. For example:

VALUES(LEFT('Hello, World!', 5)) -> 'Hello'

LENGTH(expression)

Returns the integer length of *expression*, or NULL if expression is NULL. For example:

VALUES(LENGTH('Hello, World!')) -> 13

LOCATE(substring, string [,starting_pos])

Returns the location of *substring* within *string* or zero, if the substring is not found within *string*. The optional *starting_pos* can be used to specify the starting position within *string* to begin the search. For example:

VALUES(LOCATE('World', 'Hello, World!')) -> 8

LOG(number)

Returns the natural logarithm of *number*, the same as *LN(number)*.

LOG10(number)

Returns the base 10 logarithm of *number*. For example:

VALUES(LOG10(50)) -> +1.69897000433602E+000

LONG_VARCHAR(string)

Returns a *LONG_VARCHAR* representation of *string*.

LONG_VARGRAPHIC(string)

Returns a *LONG_VARGRAPHIC* representation of *string*.

LTRIM(string)

Removes all characters inset from the left of *string*. For example:

VALUES(LTRIM(' Howdy! ')) -> 'Howdy! '

MICROSECOND(expression)

Returns the microsecond part of the time value in *expression*. For example:

VALUES(MICROSECOND(CURRENT_TIMESTAMP)) -> 252270

MIDNIGHT_SECONDS(expression)

Returns the number of seconds that have elapsed between midnight and the time value in *expression*. For example:

VALUES(MIDNIGHT_SECONDS('08:20:15')) -> 30015

MINUTE(expression)

Returns the minute part of the time value in *expression*. For example:

VALUES(MINUTE('08:20:15')) -> 20

MONTH(expression)

Returns the month part of the date value in *expression*. For example:

VALUES(MONTH('1999-04-15')) -> 4

MONTHNAME(expression)

Returns the month's name for the date value stored in *expression* using the current locale of the database. For example:

VALUES(MONTHNAME('1999-04-15')) -> 'April'

MULTIPLY_ALT(number1, number2)

Returns the product of *number1* and *number2*. The function *MULTIPLY_ALT* is a good alternative to the standard multiplication operator (*) when the numbers being multiplied have a precision greater than 31. For example:

VALUES(MULTIPLY_ALT(DECIMAL('256'), DECIMAL('256'))) -> 65536

NULLIF(expression1, expression2)

Returns NULL if *expression1* and *expression2* are equal; otherwise the function returns *expression1*. For example:

VALUES(NULLIF('1999-04-15', '2000-04-15')) -> '1999-04-15'

POSSTR(source, search)

Returns the character position of the first occurrence of *search* within *source* where the first character position is 1. For example:

VALUES(POSSTR('Hello, World!', 'World')) -> 8

QUARTER(expression)

Returns an integer ranging from 1 to 4, representing the quarter of the year that contains the date in *expression*. For example:

VALUES(QUARTER('2004-04-15')) -> 2

RADIANS(expression)

Returns the number of radians from *expression*, which is expressed in degrees. For example:

VALUES(RADIANS(180)) -> +3.14159265358979E+000

RAISE_ERROR(sqlstate, errorstring)

Raises an error upon execution. This function is useful for generating critical errors in complicated SQL statements and stored procedures. The *sqlstate* parameter is used for passing error code information back to the application executing the SQL statement, and the *errorstring* parameter is used for a custom error message. For example:

```
SELECT a.au_fname, a.au_lname,  
  
       CASE WHEN t.ytd_sales = 0 THEN 'no sales'  
            WHEN t.ytd_sales > 0 THEN 'OK'  
            ELSE RAISE_ERROR('70001', 'Sales should not be negative.')       END  
  
FROM authors a, titleauthor, titles t  
  
WHERE titleauthor.au_id = a.au_id AND  
       Titleauthor.title_id = t.title_id  
  
GROUP BY a.au_lname, a.au_fname
```

RAND([seed])

Returns a random floating-point value between 0 and 1, using *seed* as the random number generator seed.

REAL(number)

Returns a single-precision floating point representation of *number*.

REC2XML(decimal, format, rowtags, column_name [, . . .])

Returns an XML string containing XML tags, as well as column names and column data. The arguments are described in the following list:

decimal

Decimal value greater than 0, but less than or equal to 6.0, that contains the factor to expand string values to compensate for the string value expanding due to character replacements for the XML values and elements.

format

Case-sensitive string value equal to "COLATTVAL" or "COLATTVAL_XML". If column values can contain XML data that would need to be translated to produce valid XML output, then use "COLATTVAL". If the columns will not contain special XML markup characters, then use "COLATTVAL_XML". Column names containing special markup characters will still be translated (i.e., "escaped") when "COLATTVAL_XML" is chosen.

rowtags

String value to name the row element containing the column data. If *rowtags* is an empty string, then the value of **row** is used. If *rowtags* contains a string of only spaces, then the enclosing row element will be omitted in the output.

column_name

The name of a column to place into the XML result. For example:

```
SELECT REC2XML(1.3, 'COLATTVAL', 'Author', au_id, au_fname,  
    au_lname)  
FROM AUTHORS WHERE au_id = '172-32-1176'  
  
<Author>  
    <column name="AU_ID">172-32-1176</column>  
    <column name="AU_FNAME">Johnson</column>  
    <column name="AU_LNAME">White</column>  
</Author>
```

REPEAT(string, number)

Returns a character string that is composed of *string* repeated *number* times. For example:

```
VALUES( CHAR(REPEAT('Duck ', 3)) )    -> 'Duck Duck Duck'
```

REPLACE(string, search_string, replacement_string)

Returns *string* with every occurrence of *search_string* replaced with *replacement_string*. For example:

```
VALUES( REPLACE('change', 'e', 'ing') ) -> 'changing'
```

RIGHT(string, number)

Returns the *number* rightmost bytes from *string*. For example:

```
VALUES( RIGHT('Hello, World!', 6) )    -> 'World!'
```

ROUND(number[, decimal])

Returns *number* rounded to *decimal* places right of the decimal point. When *decimal* is omitted, *number* is rounded to places. Note that *decimal*, an integer, can be negative to round off digits left of the decimal point. For example:

```
VALUES( ROUND(12345.6789, 2) ) -> 12345.6800
```

RTRIM(string)

Returns *string* with all trailing whitespace characters removed. For example:

```
VALUES( RTRIM(' welcome ') )    -> ' welcome'
```

SECOND(expression)

Returns the seconds part of the time value in *expression*. For example:

```
VALUES( SECOND('08:20:15') ) -> 15
```

SIGN(number)

When *number* < 0, returns -1. When *number* = 0, returns 0. When *number* > 0, returns 1. For example:

VALUES(SIGN(-3.1415926), SIGN(0), SIGN(3.1415926))

-1.0000000000000000E+000 0 +1.0000000000000000E+000

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SINH(number)

Returns the hyperbolic sine of *number*.

SMALLINT(number)

Returns a *SMALLINT* value equivalent to *number*.

SOUNDEX(string)

Returns a character string containing the phonetic representation of *string*. This function allows words that are spelled differently but sound alike in English to be compared for equality. For example:

VALUES(SOUNDEX('thimble')) -> 'T514'

SPACE(number)

Returns a string composed of *number* spaces. For example:

VALUES(SPACE(5)) -> ' '

STDDEV({ ALL | DISTINCT } expression)

Returns the standard deviation of the values contained within *expression*. The *ALL* keyword specifies the default behavior in which all values in *expression* are used in the standard deviation. The keyword *DISTINCT* omits duplicate values in *expression* when doing the calculation. For example:

SELECT STDDEV(values) FROM NUMBERS -> 0.0642

SUBSTR(string, start [, length])

Returns a substring of *string* of *length* bytes starting at *start*. If *length* is too large to permit a valid substring, then *string* is padded with trailing spaces until a valid substring can be found. The starting position, *start*, must be an integer value in the range of 1 to the length of *string*. For example:

VALUES(SUBSTR('Hello, World!', 8, 5)) -> 'World'

TABLE_NAME(object [,schema])

Returns the unqualified name of *object* after resolving any aliases. The optional *schema* argument specifies the database schema to use in resolving the object name. For example:

VALUES(TABLE_NAME('trades')) -> 'trades'

TABLE_SCHEMA(object [,schema])

Returns the schema name of the *object* after resolving any aliases. The optional *schema* argument specifies the database schema to use in resolving the object name. For example:

VALUES(TABLE_SCHEMA('trades')) -> 'MYSHEMA'

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

SELECT TAN(3.1415) -> -0.000093

TANH(number)

Returns the hyperbolic tangent of *number*.

TIME(expression)

Returns a time value equivalent to the time in *expression*. For example:

VALUES(TIME('2003-04-15 08:20:15')) -> '08:20:15'

TIMESTAMP(expression1 [,expression2])

Returns a timestamp value created from date and time values in *expression1* and *expression2*. If only *expression1* is given, then it must contain a valid date and time value. If both arguments are given, then *expression1* must contain a valid string representation of a date and *expression2* a valid string representation of time. For example:

VALUES(TIMESTAMP('2003-04-15', '08:20:15'))

2003-04-15-08.20.15.000000

TIMESTAMP_FORMAT(string,format)

Returns a timestamp value after extracting it from *string*. The second argument, *format*, contains the timestamp format used in *string*. (For details on the timestamp format, see the *VARCHAR_FORMAT* function.) For example:

VALUES(TIMESTAMP_FORMAT('2003-4-15 08:20:15', 'YYYY-MM-DD HH24:MI:SS'))

2003-04-15-08.20.15.000000

TIMESTAMP_ISO(expression)

Returns a timestamp value from *expression*. If *expression* is a date value, then the time value will consist of zeros. If the *expression* is a time value, then the current date will be used in constructing the timestamp value. For example:

VALUES(TIMESTAMP_ISO('2003-04-15')) -> 2003-04-15-00.00.00.000000

TIMESTAMPDIFF(interval_type, tsdiff_expression)

Returns an estimated number of intervals between two timestamp values. The *interval_type* argument must be a power of 2 in the range between 1 and 256, where the meaning of the value is interpreted as in the following table:

interval_type	Timestamp difference returned
1	Fractions of a second
2	Seconds
4	Minutes
8	Hours

16	Days
32	Weeks
64	Months
128	Quarters
256	Years

The *tsdiff_expression* must be the result of subtracting two timestamp values and converting to a single *CHAR(22)* value. For example:

VALUES(TIMESTAMPDIFF(2, CHAR(CURRENT_DATE - CURRENT_TIMESTAMP)))

515

TO_CHAR(timestamp, format)

Synonym for *VARCHAR_FORMAT*.

TO_DATE(string, format)

Synonym for *TIMESTAMP_FORMAT*.

TRANSLATE(string, tostring, fromstring [,pad_char])

Returns a string equivalent to the *string* argument with each character found in *tostring* replaced with the corresponding character from *fromstring*. The optional *pad_char* argument can contain a single-byte character that will be used to pad *tostring* if it is shorter than *fromstring*. If the same character appears more than once in *fromstring*, then only the first occurrence will count and all following occurrences will be ignored. For example:

VALUES(TRANSLATE('123,456,789.45', ',', '.')) -> '123.456.789,45'

TRUNCATE(number1, number2) or *TRUNC(number1, number2)*

Returns *number1* with all digits to the right of the decimal point by *number2* places replaced with zeros. If *number2* is negative, then the truncation occurs to the left of the decimal place. For example:

VALUES(TRUNCATE(DEC('123.456'), -1)) -> 120.

TYPE_ID(expression)

Returns the internal type identifier to the user-defined structured type instance stored in *expression*. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

TYPE_NAME(expression)

Returns the unqualified name to the user-defined structured type instance stored in *expression*. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

TYPE_SCHEMA(expression)

Returns the schema name to the user-defined structured type instance stored in *expression*. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

VALUE(expression [,...])

Returns the first argument that is not NULL. For example:

VALUES(VALUE('Hello!', 5)) -> 'Hello!'

VAR or **VARIANCE**({ *ALL* | *DISTINCT* } *expression*)

Returns the variance of the values contained within *expression*. The *ALL* keyword specifies the default behavior in which all values in *expression* are used in the calculation. The keyword *DISTINCT* omits duplicate values in *expression* when doing the calculation. For example:

SELECT VARIANCE(values) FROM NUMBERS -> 987244882.22

VARCHAR(*expression* [, *length*])

Returns a varying-length character representation of *expression*, where *length* is the length attribute for the varying-length character result. If *length* is omitted, then the length of the result is the length of the expression.

VARCHAR_FORMAT(*expression*, *format*)

Returns a string representation of the timestamp value in *expression* in the *format* provided in the *format* argument. Listed below are the available specifiers for *format* and their meanings.

YYYY

Four-digit year

MM

Two-digit month (01-12)

DD

Two-digit day of month (01, 02, ...)

HH24

Two-digit hour of day (00-24)

MI

Two-digit minute (00-59)

SS

Two-digit second (00-59)

For example:

VALUES(VARCHAR_FORMAT(CURRENT_TIMESTAMP, 'YYYY-MM-DD HH24:MI:SS'))

2003-06-24 09:37:45

VARGRAPHIC(*expression* [, *length*])

Returns a varying-length graphic string representation of *expression*, where *length* is the length attribute for the varying-length graphic string result. If *length* is omitted, then the length of the result is the length of the expression.

WEEK(*expression*)

Returns the week of the year for the date value contained in *expression*. The range for the result is 1-54 and the week starts with Sunday. For example:

VALUES(WEEK('2003-04-15')) -> 16

WEEK_ISO(expression)

Returns the week of the year for the date value contained in *expression*. The range for the result is 1-53, and the week starts with Monday. The first week of the year always contains January 4th. For example:

VALUES(WEEK_ISO('2003-04-15')) -> 16

YEAR(expression)

Returns the year part of the date value in *expression*. For example:

VALUES(YEAR('2004-04-15')) -> 2004

4.5.2 MySQL-Supported Functions

This section provides an alphabetical listing of MySQL-supported functions, with examples and corresponding results.

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

SELECT ACOS(0) -> 1.570796

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

SELECT ASCII('x') -> 120

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

SELECT ASIN(0) -> 0.000000

ATAN(number)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

SELECT ATAN(3.1415) -> 1.262619

ATAN2(x,y)

Returns the arctangent of the two variables *x* and *y*. *ATAN2(x,y)* is similar to *ATAN(y/x)*, with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

ATAN2(3.1415, 1) -> 1.262619

BENCHMARK(*count*,*expr*)

Executes the expression (*expr*) *count* times. The result value is always 0. For example:

BENCHMARK(1000000,ATAN2(3.1415, 1)) -> 0

BIN(*number*)

Returns a string containing the binary value of *n*, where *n* is a *BIGINT* number.

BINARY *string*

Casts *string* to a binary string.

BIT_AND(*expr*)

An aggregate function that returns the bitwise *AND* of all bits in *expr*. The calculation is performed with 64-bit (*BIGINT*) precision. The value of -1 is returned when no matching rows are found. For example:

BIT_AND(mycolumn) -> 0

BIT_COUNT(*number*)

Returns the number of bits that are set in *number*. For example:

BIT_COUNT(5) -> 2

BIT_OR(*expr*)

An aggregate function that returns the bitwise *OR* of all bits in *expr*. The calculation is performed with 64-bit (*BIGINT*) precision. The value of zero is returned when no matching rows are found. For example:

BIT_OR(mycolumn) -> 1

CHAR(*number* [,...])

Returns a string consisting of the characters given by the ASCII code values in the arguments. Any NULL values are ignored. For example:

CHAR(120,121,122) -> 'xyz'

COALESCE(*list*)

Returns the first non-NULL element in the list. For example:

COALESCE(NULL, 1, 2) -> 1

COMPRESS(*string*)

Returns a compressed version of *string*.

CONCAT_WS(*separator*, *str1*, *str2*[,...])

A special form of **CONCAT**() that inserts *separator* between every pair of string arguments concatenated. If *separator* is NULL, then the result is NULL. For example:

CONCAT_WS(' ', au_lname, au_fname) -> 'Jefferson, Thomas'

CONNECTION_ID()

Returns the connection ID for the connection. Every connection has its own unique ID. For example:

CONNECTION_ID() -> 305102

CONV(*number,from_base,to_base*)

Returns a string representation of the number *number*, converted from base *from_base* to base *to_base*. If any argument is NULL, then the result is NULL. For example:

CONV(12,10,2) -> 1100

COS(*number*)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

CURDATE()

Returns today's date as a value in YYYY-MM-DD or YYYYMMDD format, depending on whether the function is used in a string or numeric context. For example:

CURDATE() -> '2003-06-24'

CURTIME()

Returns the current time as a value in HH:MM:SS or HHMMSS format, depending on whether the function is used in a string or numeric context. For example:

CURTIME() -> '20:40:20'

DATABASE()

Returns the current database name. For example:

DATABASE() -> 'PUBS'

DATE_ADD(*date,INTERVAL expr type*)

DATE_SUB(*date,INTERVAL expr type*)

ADDDATE(*date,INTERVAL expr type*)

SUBDATE(*date,INTERVAL expr type*)

These functions perform date arithmetic calculations. *ADDDATE()* and *SUBDATE()* are synonyms for *DATE_ADD()* and *DATE_SUB()*. *DATE_ADD()* returns the result of adding the *INTERVAL* to the *date* expression. *DATE_SUB()* is the result of subtracting the *INTERVAL* from the *date* expression. For example:

DATE_ADD('1999-04-15', INTERVAL 1 DAY) -> '1999-04-16'

DATE_SUB('1999-04-15', INTERVAL 1 DAY) -> '1999-04-14'

DATE_FORMAT(date, format)

Formats the *date* value according to the *format* string. Listed below are the available specifiers for *format* and their meanings.

%a

Abbreviation of the day (Sun-Sat)

%b

Abbreviation of the month (Jan-Dec)

%c

Month number (1-12)

%D

Day of month with a suffix (1st, 2nd, 3rd, ...)

%d

Two digit day of month (01, 02, ...)

%e

Day of month (1, 2, 3, ...)

%H

Hour (00-23)

%h

Hour (01-12)

%i

Minutes (00-59)

%I

Hour (01-12)

%j

Day of year (001-366)

%k

Hour (0-23)

%l

Hour (1-12)

%M

Full month name (January-December)

%m

Month (01-12)

%p

A.M. or P.M.

%r

12-hour time (hh:mm:ss A.M or P.M)

%S, %s

Seconds (00-59)

%T

24-hour time (hh:mm:ss)

%U

Week number (00-53, Sunday being the first day of the week)

%u

Week number (00-53, Monday being the first day of the week)

%V

Week number (01-53, Sunday being the first day of the week)

%v

Week number (01-53, Monday being the first day of the week)

%W

Name of the day (Sunday-Saturday)

%w

Day of the week (0 = Sunday, 6 = Saturday)

%X

Four-digit year with Sunday being the first day of the week

`%x`

Four-digit year with Monday being the first day of the week

`%Y`

Four-digit year

`%y`

Two-digit year

`%%`

Literal "%"

For example:

`DATE_FORMAT('1999-04-15', '%M-%D-%Y')` -> 'April-15th-1999'

DAYNAME(date)

Returns the name of the weekday for *date*. For example:

`DAYNAME('1999-04-15')` -> 'Thursday'

DAYOFMONTH(date)

Returns the day of the month for *date*, in the range 1 to 31. For example:

`DAYOFMONTH('1999-04-15')` -> 15

DAYOFWEEK(date)

Returns the weekday index for *date* (1 = Sunday, 2 = Monday, . . . 7 = Saturday). For example:

`DAYOFWEEK('1999-04-15')` -> 5

DAYOFYEAR(date)

Returns the day of the year for *date*, in the range 1 to 366. For example:

`DAYOFYEAR('1999-04-15')` -> 105

DECODE(crypt_str, pass_str)

Decrypts the encrypted string *crypt_str* using *pass_str* as the password; *crypt_str* should be a string returned from *ENCODE()*. For example:

`DECODE(ENCODE('foo','bar'),'bar')` -> 'foo'

DEGREES(number)

Returns the argument *number* converted from radians to degrees. For example:

`DEGREES(3.1415926)` -> 179.99999692953

ELT(*n, str1, str2, str3*[,...n])

Returns *str1* if *n* = 1, *str2* if *n* = 2, and so on. If *n* is less than 1 or greater than the number of arguments, then this function returns NULL. **ELT**() is the complement of **FIELD**(). For example:

ELT(1, 'Hi', 'There') -> 'Hi'

ELT(2, 'Hi', 'There') -> 'There'

EMPTY_BLOB and **EMPTY_CLOB**

Returns empty **LOB** locators, which you can use to initialize **BLOB** and **CLOB** values that you wish to create as part of an **INSERT** or **UPDATE** statement.

ENCODE(*str, pass_str*)

Encrypts *str* using *pass_str* as the password. To decrypt the result, use **DECODE**(). The result is a binary string the same length as the string. For example:

DECODE(**ENCODE**('foo','bar'),'bar') -> 'foo'

ENCRYPT(*str*[, *salt*])

Encrypts *str* using the Unix **crypt**() system call. The *salt* argument should be a string with two characters. For example:

ENCRYPT('password') -> 'ZB7yqPUHvNnmo'

EXPORT_SET(*bits, on, off*[, *separator* [, *number_of_bits*]])

Returns a string where every bit set in *bits* gets an *on* string and every unset bit gets an *off* string. Each string is separated with *separator*; the default is a comma (,). Only *number_of_bits* of bits is used; the default is 64. For example:

EXPORT_SET(4, 'T', 'F')

F,F,T,F,

F,F

FIELD(*str, str1, str2, str3*[, ...])

Returns the index of *str* in the given string arguments. Returns 0 if *str* is not found. **FIELD**() is the complement of **ELT**(). For example:

FIELD('GOOSE', 'DUCK', 'DUCK', 'GOOSE', 'DUCK') -> 3

FIND_IN_SET(*str, strlist*)

Returns the index of *str* within the *strlist*, where *strlist* is a list of strings separated by commas. This function is equivalent to calling **FIELD**(*str*, **CONCAT_WS**('', *str1, str2, str3*, [...])). For example:

FIND_IN_SET('b', 'a,b,c,d') -> 2

FORMAT(*number, decimals*)

Formats the number *number* to a format like #,###,###.##, rounded to *decimals* decimals. If *decimals* is 0, the result has no decimal point or fractional part. For example:

FORMAT(12345.2132,2) -> 12,345.21

FORMAT(12345.2132,0) -> 12,345

FROM_DAYS(number)

Given a day number *number*, returns a *DATE* value. This function should not be used for values that precede the advent of the Gregorian calendar (1582), due to the days lost when the calendar was changed. For example:

FROM_DAYS(888888) -> 2433-09-10

FROM_UNIXTIME(unix_timestamp)

Returns a representation of the *unix_timestamp* argument as a value in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. For example:

FROM_UNIXTIME(888123892) -> 1998-02-21 21:04:52

FROM_UNIXTIME(unix_timestamp, format)

Returns a string representation of the *unix_timestamp*, formatted according to the *format* string. *format* may contain the same specifiers as those listed in the entry for the *DATE_FORMAT()* function. For example:

FROM_UNIXTIME(888123892, '%Y %D %M') -> '1998 21st February'

GET_LOCK(str, timeout)

Tries to obtain a lock with a name given by the string *str*, with a time-out of *timeout* seconds. Returns 1 if the lock is obtained successfully or NULL if an error occurs or the attempt to acquire the lock times out. For example:

GET_LOCK('lochness',10) -> 1

GREATEST(x,y[,...])

Returns the largest argument. For example:

GREATEST(8,2,4) -> 8

HEX(number)

Returns a string representation of the hexadecimal value of *number*. This is equivalent to *CONV(number,10,16)*. For example:

HEX(255) -> FF

HOUR(time)

Returns the hour for *time*, in the range 0 to 23. For example:

HOUR('08:20:15') -> 8

IF(expr1, expr2, expr3)

Returns *expr2* if *expr1* is TRUE, otherwise returns *expr3*. For example:

IF(1,'yes','no') -> 'yes'

IF(0,'yes','no') -> 'no'

IFNULL(expr1, expr2)

Returns *expr1* if *expr1* is not NULL, otherwise returns *expr2*. For example:

IFNULL(0,'NULL') -> 0

IFNULL(NULL,'NULL') -> 'NULL'

INSERT(str, pos, len, newstr)

Returns the string *str* with *newstr* inserted at character position *pos* for length *len*. For example:

INSERT('paper',2,3,'ea') -> 'pear'

INSTR(str,substr)

Returns the position of the first occurrence of the substring *substr* in the string *str*. For example:

INSTR('ducks','c') -> 3

INTERVAL(num1,num2,num3,num4[,...n])

Returns 0 if *num1* < *num2*, 1 if *num1* < *num3*, and so on. It is required that *num2* < *num3* < *num4* < . . . < *numN*. For example:

INTERVAL(5,1,6) -> 1

INTERVAL(5,2,3,7,9) -> 2

IS_FREE_LOCK(lock)

Returns 1 if *lock* is free and zero if the lock is currently in use. The function may return NULL on error conditions. For example:

IS_FREE_LOCK('lochness') -> 0

ISNULL(expr)

If *expr* is NULL, *ISNULL()* returns 1; otherwise it returns 0. For example:

ISNULL(1) -> 0

ISNULL(NULL) -> 1

LAST_INSERT_ID([expr])

Returns the last automatically generated value that was inserted into an *AUTO_INCREMENT* column. For example:

LAST_INSERT_ID() -> 0

LCASE(str)

Synonym for *lower(str)*. For example:

LCASE('DUCK') -> 'duck'

LEAST(X,Y[,...n])

With two or more arguments, returns the smallest (minimum-valued) argument. For example:

LEAST(10,5,3,7) -> 3

LEFT(str,len)

Returns the leftmost *len* characters from the string *str*. For example:

LEFT('Ducks', 4) -> 'Duck'

LENGTH(str)

These functions return the length of the string *str*. For example:

LENGTH('DUCK') -> 4

LOAD_FILE(file_name)

Reads the file and returns the file contents as a string. The file must be on the server, and the user must specify the full pathname to the file and have access to the file.

LOCATE(substr,str), POSITION(substr IN str)

Returns the position of the first occurrence of the substring *substr* in the string *str*. Returns zero if *substr* is not in *str*. *LOCATE* is a synonym for the standard *POSITION(substr IN str)*. For example:

LOCATE('al','Donald') -> 4

POSITION('al' IN 'Donald') -> 4

LOCATE(substr,str,pos)

Returns the position of the first occurrence of the substring *substr* in the string *str*, starting at position *pos*; returns zero if *substr* is not in *str*. For example:

LOCATE('World', 'Hello, World!') -> 8

LOG(X)

Returns the natural logarithm of *X*. For example:

LOG(50) -> 3.912023

LOG2(X)

Returns the base-2 logarithm of *X*. For example:

LOG2(50) -> 5.64386

LOG10(X)

Returns the base-10 logarithm of *X*. For example:

LOG10(50) -> 1.698970

LPAD(str,len,padstr)

Returns the string *str*, left-padded with the string *padstr* until *str* is *len* characters long. For example:

LPAD('ucks',6,'d') -> 'dducks'

LTRIM(str)

Returns the string *str* with leading-space characters removed. For example:

LTRIM(' Howdy! ') -> 'Howdy! '

MAKE_SET(bits, str1, str2[, ...n])

Returns a *set* (a string containing substrings separated by commas) consisting of the string arguments that have the corresponding bit in *bits* set; *str1* corresponds to bit 0, *str2* to bit 1, etc. NULL strings in *str1*, *str2*, ... are not appended to the result. For example:

MAKE_SET(1 | 4,'hello','nice','world') -> 'hello,world'

MD5(string)

Calculates an *MD5* checksum for the *string*. Value is returned as a 32-digit-long hex number. For example:

MD5('somestring') -> 1f129c42de5e4f043cbd88ff6360486f

MINUTE(time)

Returns the minute for *time*, in the range to 59. For example:

MINUTE('08:20:15') -> 20

MONTH(date)

Returns the month for *date*, in the range 1 to 12. For example:

MONTH('1999-04-15') -> 4

MONTHNAME(date)

Returns the name of the month for *date*. For example:

MONTHNAME('1999-04-15') -> 'April'

NOW() , SYSDATE()

Returns the current date and time as a value in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. For example:

NOW() -> 2003-06-24 20:40:24

SYSDATE() -> 2003-06-24 20:40:24

CURRENT_TIMESTAMP -> 2003-06-24 20:40:24

NULLIF(expr1, expr2)

Returns NULL if *expr1* is equal to *expr2*; otherwise returns *expr1*. For example:

NULLIF(2,29) -> 2

NULLIF(29,29) -> NULL

OCT(n)

Returns an octal value equivalent of *n*, where *n* is a number. This is equivalent to *CONV(N,10,8)*. Returns NULL if *n* is NULL. For example:

OCT(255) -> 377

ORD(str)

Returns the character ordinal of the multibyte character string *str*. The value is calculated using the following formula: ((first byte ASCII code)*256+(second byte ASCII code)*256*256)(third byte ASCII code)*256*256*256[,...]. If *str* isn't a multibyte character, then this function returns the same value as the *ASCII()* function. For example:

ORD('29') -> 50

PASSWORD(str)

Calculates a password string from the plain-text password *str*. This is the function that is used for encrypting MySQL passwords. For example:

PASSWORD('password') -> 5d2e19393cc5ef67

PERIOD_ADD(period, months)

Adds the number of months found in *months* to the period in *period* (in the format Y YMM or YYYYMM). Returns a value in the format YYYYMM. For example:

PERIOD_ADD(9902,3) -> 199905

PERIOD_DIFF(period1, period2)

Returns the number of months between *period1* and *period2*. *period1* and *period2* should be in the format Y YMM or YYYYMM. For example:

PERIOD_DIFF(9902,9905) -> -3

PI()

Returns the value of π . For example:

PI() -> 3.141593

POW(X, Y), POWER(X, Y)

Returns the value of *X* raised to the power of *Y*. For example:

POW(2, 8) -> 256.000000

QUARTER(date)

Returns the quarter of the year for *date*, in the range 1 to 4. For example:

QUARTER('1999-04-15') -> 2

RADIANS(X)

Returns the argument *X*, converted from degrees to radians. For example:

RADIANS(180) -> 3.1415926535898

RAND() , RAND(N)

Returns a random floating-point value in the range 0 to 1.0. If an integer argument *N* is specified, it is used as the seed value. For example:

RAND() -> 0.29588872501244

RELEASE_LOCK(str)

Releases the lock named by the string *str* that was obtained with *GET_LOCK()*. Returns 1 if the lock is released or NULL if the named lock doesn't exist or isn't locked by this thread (in which case the lock is not released). For example:

RELEASE_LOCK('lochness') -> 1

REPEAT(str, count)

Returns a string consisting of the string *str* repeated *count* times. For example:

REPEAT('Duck', 3) -> 'DuckDuckDuck'

REPLACE(str, from_str, to_str)

Returns the string *str* with all occurrences of the string *from_str* replaced by the string *to_str*. For example:

REPLACE('change', 'e', 'ing') -> 'changing'

REVERSE(str)

Returns the string *str* reversed. For example:

REVERSE('STOP') -> 'POTS'

RIGHT(str, ten)

Returns the rightmost 10 characters from the string *str*. For example:

RIGHT('Hello, World!', 6) -> 'World!'

ROUND(X[,D])

Returns the argument *X*, rounded to a number with *D* decimals. If *D* is 0, the result has no decimal point or fractional part. For example:

ROUND(12345.6789, 2) -> 12345.68

RPAD(str,len,padstr)

Returns the string *str*, right-padded with the string *padstr* until *str* is *len* characters long. For example:

RPAD('duck',6,'s') -> 'duckss'

RTRIM(str)

Returns the string *str* with trailing space characters removed. For example:

RTRIM(' welcome ') -> 'welcome '

SEC_TO_TIME(seconds)

Returns the *seconds* argument, converted to hours, minutes, and seconds, as a value in HH:MM:SS or HHMMSS format, depending on whether the function is used in a string or numeric context. For example:

SEC_TO_TIME(256) -> 00:04:16

SECOND(time)

Returns the second for *time*, in the range to 59. For example:

SECOND('08:20:15') -> 15

SHA(X) or *SHA1(X)*

Returns a SHA1 160-bit checksum for *X*. For example:

SHA('abc') -> 'a9993e364706816aba3e25717850c26c9cd0d89d'

SIGN(X)

Returns the sign of the argument as -1, 0, or 1, depending on whether *X* is negative, zero, or positive. For example:

SIGN(-3.1415926) -> -1

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SOUNDEX(str)

Returns a soundex string from *str*. For example:

SOUNDEX('thimble') -> 'T514'

expr1 SOUNDS LIKE expr2

Synonymous with the expression:

SOUNDEX(*expr1*) = SOUNDEX(*expr2*)

SPACE(n)

Returns a string consisting of *n* space characters. For example:

SPACE(5) -> ' '

STD(expr), STDDEV(expr)

Returns the standard deviation of *expr*. The *STDDEV()* form of this function is provided for Oracle compatibility. For example:

STD(5) -> NULL

STRCMP(expr1, expr2)

STRCMP() returns zero, as marked if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise. For example:

STRCMP('DUCKY', 'DUCK') -> 1

STRCMP('DUCK', 'DUCK') -> 0

SUBSTRING(str, pos, len), MID(str, pos, len)

Returns a substring len characters long from string *str*, starting at position *pos*. These functions are synonyms for the ANSI SQL92 function *SUBSTRING(str FROM pos FOR len)*. For example:

SUBSTRING('Hello, World!', 8, 10) -> 'World!'

SUBSTRING('Hello, World!' FROM 8 FOR 10) -> 'World!'

SUBSTRING(str, pos), SUBSTRING(str FROM pos)

Returns a substring from string *str* starting at position *pos*. For example:

SUBSTRING('Hello, World!', 8) -> 'World!'

SUBSTRING('Hello, World!' FROM 8) -> 'World!'

SUBSTRING_INDEX(str, delim, count)

Returns the substring of *str* after *count* occurrences of the delimiter *delim*. For example:

SUBSTRING_INDEX('www.mysql.com', '.', 2) -> 'www.mysql'

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

SELECT TAN(3.1415) -> -0.000093

TIME_FORMAT(time, format)

This is used like *DATE_FORMAT()*, but the *format* string may contain only those format specifiers that handle hours, minutes, and seconds. Other specifiers produce a NULL value or 0. See *DATE_FORMAT* for the format specifiers available. For example:

TIME_FORMAT('2003-04-15 08:20:15', '%r') -> 08:20:15 AM

TIME_TO_SEC(time)

Returns the *time* argument, converted to seconds. For example:

TIME_TO_SEC('08:20:15') -> 30015

TO_DAYS(date)

Given a *date*, returns a day number (the number of days since the year 0). For example:

TO_DAYS('1999-04-15') -> 730224

TRUNCATE(X,D)

Returns the number *X*, truncated to *D* decimals. If *D* is 0, the result has no decimal point or fractional part. For example:

TRUNCATE('123.456', 2) -> '123.45'

TRUNCATE('123.456', 0) -> '123'

TRUNCATE('123.456', -1) -> '120'

UCASE(str)

Synonym for *UPPER(str)*. For example:

UCASE('duck') -> 'DUCK'

UNCOMPRESS(string)

Returns a compressed version of *string*.

UNHEX(str)

Returns a binary string constructed from hex characters in *str*.

UNIX_TIMESTAMP() , *UNIX_TIMESTAMP(date)*

If called with no argument, returns a Unix timestamp (seconds since 1970-01-01 00:00:00 GMT). If *UNIX_TIMESTAMP()* is called with a *date* argument, it returns the value of the argument as seconds since 1970-01-01 00:00:00 GMT. For example:

UNIX_TIMESTAMP() -> 1056512427

UNIX_TIMESTAMP('1999-04-15') -> 924159600

USER() , *SYSTEM_USER()* , *SESSION_USER()*

These functions return the current MySQL username. For example:

USER() -> 'login@machine.com'

SYSTEM_USER() -> 'login@machine.com'

SESSION_USER() -> 'login@machine.com'

VERSION()

Returns a string indicating the MySQL server version. For example:

VERSION() -> '4.0.12-standard'

WEEK(date), WEEK(date, first)

With a single argument, returns the week for *date*, in the range 1 to 53. (The beginning of a week 53 is possible during some years.) The two-argument form of *WEEK()* allows the user to specify whether the week starts on Sunday (0) or Monday (1). For example:

WEEK('1999-04-15') -> 15

WEEKDAY(date)

Returns the weekday index for *date* (0 = Monday, 1 = Tuesday, . . . 6 = Sunday). For example:

WEEKDAY('1999-04-15') -> 3

YEAR(date)

Returns the year for *date*, in the range 1000 to 9999. For example:

YEAR('1999-04-15') -> 1999

YEARWEEK(date), YEARWEEK(date, first)

Returns the year and week for *date*. The second argument works exactly like the second argument to *WEEK()*. Note that the year may be different from the year in the *date* argument for the first and the last week of the year. For example:

YEARWEEK('1999-04-15') -> 199915

4.5.3 Oracle-Supported Functions

This section provides an alphabetical listing of the SQL functions specific to Oracle, with examples and corresponding results.

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to 1 and is expressed in radians. For example:

SELECT ACOS(0) FROM DUAL -> 1.570796

ADD_MONTHS(date, int)

Returns the *date* plus *int* months. For example:

SELECT ADD_MONTHS('15-APR-1999', 3) FROM DUAL -> 15-JUL-99

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

SELECT ASCII('x') FROM DUAL -> 120

ASCIISTR(*text*)

Converts *text* from any character set into an ASCII equivalent. Characters in *text* that have no equivalent in ASCII will be replaced with the string \XXXX, where XXXX represents the UTF-16 code unit. For example:

```
SELECT ASCIISTR('ÄBC') FROM DUAL -> '\00C4BC'
```

ASIN(*number*)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from -/2 to /2 and is expressed in radians. For example:

```
SELECT ASIN( 0 ) FROM DUAL -> 0.000000
```

ATAN(*number*)

Returns the arctangent of *number*. The resulting value ranges from -/2 to /2 and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) FROM DUAL -> 1.262619
```

ATAN2(*number, nbr*)

Returns the arctangent of *number* and *nbr*. The values for *number* and *nbr* are not restricted, but the results range from - to and are expressed in radians. ATAN2(*x,y*) is similar to ATAN(*y/x*), with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
SELECT ATAN2(3.1415, 1) FROM DUAL -> 1.26261873
```

BFILENAME(*directory, filename*)

Returns a *BFILE* locator associated with a physical *LOB* binary file on the server's filesystem in *directory* with the name *filename*.

BIN_TO_NUM(*expr[,...n]*)

Returns a decimal number equivalent of the binary bit vector contained in the *expr* arguments. For example:

```
SELECT BIN_TO_NUM(1,0,1) FROM DUAL -> 5
```

BITAND(*integer1, integer2*)

Returns the bitwise *AND* of the two integer arguments. For example:

```
SELECT BITAND(101, 2) FROM DUAL -> 0
```

```
SELECT BITAND(column1, 1) FROM DUAL -> 1
```

CARDINALITY(*nested_table*)

Returns the number of elements (cardinality) of the *nested_table*. If the *nested_table* is empty, NULL will be returned. For example:

```
SELECT CARDINALITY(mytable) FROM DUAL -> 6
```

CHARTOROWID(*char*)

Converts a value from a character datatype (*CHAR* or *VARCHAR2* datatype) to a *ROWID* datatype.

CHR(number [USING NCHAR_CS])

Returns the character having the binary equivalent to *number* in either the database character set (if *USING NCHAR_CS* is not included) or the national character set (if *USING NCHAR_CS* is included).

COALESCE(list)

Returns the first non-NULL element in the list. For example:

SELECT COALESCE(NULL, 1, 2) FROM DUAL -> 1

COLLECT (column)

Creates for each group a nested table consisting of all values in a column. This is an aggregate function.

COMPOSE(string)

Returns *string* as a fully normalized UNICODE string.

CONCAT(string1, string2)

Returns *string1* concatenated with *string2*. *CONCAT* is equivalent to the concatenation operator (||). For example:

SELECT CONCAT(au_lname, au_fname) FROM AUTHORS -> 'JeffersonThomas'

CONVERT(char_value, target_char_set, source_char_set)

Converts a character string from one character set to another; returns *char_value* in the *target_char_set* after converting *char_value* from the *source_char_set*.

CORR_K(expr1, expr2 [,return_type])

CORR_S(expr1, expr2 [,return_type])

CORR_K returns Kendall's tau-b correlation coefficient, and *CORR_S* returns Spearman's rho correlation coefficient of a set of numbered pairs (*expr1* and *expr2*). The *return_type* argument, a *VARCHAR2*, can be omitted or one of the following values: 'COEFFICIENT', 'ONE_SIDED_SIG', 'TWO_SIDED_SIG'. The value of 'COEFFICIENT', which specifies the default behavior, returns the coefficient of the correlation. The values of 'ONE_SIDED_SIG' and 'TWO_SIDED_SIG' will return the one- and two-tailed significance of the correlation, respectively.

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) FROM DUAL -> 1.000000

COSH(number)

Returns the hyperbolic cosine of number. For example:

SELECT COSH(180) FROM DUAL -> 7.4469E+77

SELECT COT(3.1415) FROM DUAL -> -10792.88993953

COT(number)

Returns the cotangent of *number*. For example:

```
SELECT COT( 3.1415 ) -> -10792.88993953
```

CV([*dimension_column*])

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns the current value of the *dimension_column*. *CV* can only be used in the righthand side of a rule, since it returns the value of the *dimension_column* from the left-hand side of the same rule.

DBTIMEZONE

Returns the time zone offset from UTC time for the database server. For example:

```
SELECT DBTIMEZONE FROM DUAL -> +00:00
```

DECODE(*expr*, *search* , *result* [, *search*, *result* [,...*n*]] [,*default*])

Compares *expr* to the *search* value; if *expr* is equal to a *search*, it returns the result. For example:

```
DECODE ('B','A',1,'B',2,...'Z',26,'?') -> 2
```

Without a match, *DECODE* returns *default*, or NULL if *default* is omitted. Refer to Oracle documentation for more details. Consider using *CASE* instead, as *CASE* is part of the ANSI/ISO SQL standard.

DECOMPOSE(*string* [{CANONICAL | COMPATIBILITY}])

Returns *string* decomposed into UNICODE code-points. The second argument specifies the type of decomposition performed. *CANONICAL*, which specifies the default behavior, allows the original UNICODE string to be recomposed.

DEPTH(*number*)

Returns the depth of the path specified by the *UNDER_PATH* condition in an XML query. See the Oracle SQL Reference for more information.

DEREF(*expression*)

Returns the object referenced by *expression*, where *expression* must return a *REF* to an object.

DUMP(*expression* [,*return_format* [, *starting_at* [,*length*]]])

Returns a *VARCHAR2* value containing a datatype code, *length* in bytes, and internal representation of *expression*. The resulting value is returned in the format of *return_format*. For example:

```
SELECT DUMP('abc', 1016) FROM DUAL
```

```
Typ=96 Len=3 CharacterSet=AL32UTF8: 61,62,63
```

EMPTY_BLOB() , *EMPTY_CLOB*()

Returns an empty *LOB* locator that can be used to initialize a *LOB* variable. It can also be used to initialize a *LOB* column or attribute to empty in an *INSERT* or *UPDATE* statement.

EXISTSNODE(*instance*, *xpath* [, *namespace*])

Returns 1 if applying the XPath query in *xpath* would return any nodes from *instance*; otherwise, returns 0. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference.

EXTRACT(*instance*, *xpath* [, *namespace*])

Returns the XML nodes from *instance* returned by running the XPath query contained in the *xpath* parameter. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

```
SELECT EXTRACT( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),  
                '/foo/bar' ) from DUAL
```

```
<bar>Hello, World!</bar>
```

EXTRACTVALUE(*instance*, *xpath* [, *namespace*])

Returns the value from an XML node returned by running the XPath query contained in the *xpath* parameter. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

```
SELECT EXTRACTVALUE( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),  
                    '/foo/bar' ) from DUAL
```

```
Hello, World!
```

FIRST

An aggregate function that returns a specified value from the row that ranks first, given the order specified in the *ORDER BY* clause. Syntax:

```
aggregate(aexpr) KEEP (DENSE_RANK FIRST ORDER BY expr [,...n])
```

where the syntax of *expr* is:

```
expr := [ASC|DESC] [NULLS {FIRST|LAST}]
```

The first ranking row following the order specified by *expr* will be used in the aggregate function *aggregate*. The *aexpr* is the expression passed to the aggregate function. For example:

```
SELECT MAX(c1) KEEP (DENSE_RANK FIRST ORDER BY c2) FROM FIVE_NUMS
```

```
1
```

FIRST_VALUE(*expression* IGNORE NULLS) *OVER* (*window_clause*)

Returns the first value in an ordered set of values. *FIRST_VALUE* is an analytic function. See the [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

```
SELECT FIRST_VALUE(col1) OVER ( ) FROM NUMS
```

```
1
```

```
1
```

```
1
```

```
1
```

FROM_TZ(*timestamp*, *timezone*)

Returns *timestamp* converted to a *TIMESTAMP WITH TIME ZONE* value, where *timestamp* is a *TIMESTAMP* value and *timezone* is a string in the TZH:TZM format. For example:

```
SELECT FROM_TZ(TIMESTAMP '2004-04-15 23:59:59', '8:00') FROM DUAL
```


'15-APR-04 11.59.59 PM +08:00'

GREATEST(expression [,...n])

Returns the greatest of the list of *expressions*. All *expressions* after the first are implicitly converted to the datatype of the first expression before the comparison. For example:

SELECT GREATEST(8,2,4) FROM DUAL -> 8

GROUP_ID()

Returns a positive value for each duplicate group returned by a query containing a *GROUP BY* clause. This function is useful in filtering out duplicate groups created when using *CUBE*, *ROLLUP*, or other *GROUP BY* extension (see *GROUPING*.)

GROUPING(column_name)

Returns 1 when a row is added by *CUBE*, *ROLLUP*, or other *GROUP BY* extensions; otherwise returns 0. For example:

**SELECT royalty, SUM(advance) 'total advance',
GROUPING(royalty) 'grp'
FROM titles
GROUP BY royalty WITH ROLLUP**

royalty	total advance	grp
-----	-----	---
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

GROUPING_ID(column_name1 [, column_name2 ,...])

Returns the base-10 number that is equal to the binary value constructed by concatenating the *GROUPING* values on each of the parameters. *GROUPING_ID* is useful when returning a query containing multiple levels of aggregation created by *GROUP BY* expressions. Consider using the *GROUPING_ID* function instead of multiple *GROUPING* functions within one query. This function is a shorthand equivalent of:

BIN_TO_NUM(GROUPING(column_name1) [, GROUPING(column_name2) ,...])

HEXTORAW(string)

Converts *string* containing hexadecimal digits into a raw value. For example:

SELECT HEXTORAW('0FE') FROM DUAL -> '00FE'

INITCAP(string)

Returns *string*, with the first letter of each word in uppercase and all other letters in lowercase. For example:

```
SELECT INITCAP('thomas jefferson') FROM DUAL -> 'Thomas Jefferson'
```

INSTR(*string1*, *string2* [, *start_at* [, *occurrence*]])

Returns the position of *string2* within *string1*. *INSTR* searches *string1* with a starting position of *start_at* (an integer) looking for the specified *occurrence* of *string2*. For example:

```
SELECT INSTR('foobar', 'o', 1, 1) FROM DUAL -> 2
```

Use *INSTRB* for bytes, *INSTRC* for UNICODE complete characters, *INSTR2* for UNICODE UCS2 code points, and *INSTR4* for UNICODE UCS4 code points.

ITERATION_NUMBER

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns the number of times the rules within the *MODEL* clause have been executed while processing the query.

LAG(*expression* [, *offset*] [, *default*]) *OVER* (*window_clause*)

An analytic function that provides access to more than one row of a table at the same time without a self join. *LAG* provides a "lagging" value in the result set that lags *offset* rows behind the current row. The *default* value is used for the first *offset* rows in the result set, since the "lagging" value is undefined for these rows. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

```
SELECT c1, LAG(c1, 2, 0) OVER (ORDER BY c1) FROM FIVE_NUMS
```

```
1    0
2    0
3    1
4    2
5    3
```

LAST

Returns the row that ranks last given the order specified in the *ORDER BY* clause. The syntax is:

```
aggregate(aexpr) KEEP (DENSE_RANK LAST ORDER BY expr [,...n])
```

where the syntax of *expr* is:

```
expr := [ASC|DESC] [NULLS {FIRST|LAST}]
```

The last ranking row following the order specified by *expr* will be used in the aggregate function *aggregate*. The *aexpr* is the expression passed to the aggregate function. For example:

```
SELECT MIN(c1) KEEP (DENSE_RANK LAST ORDER BY c1) FROM FIVE_NUMS
```

```
5
```

LAST_DAY(*date*)

Returns the date of the last day of the month that contains *date*. For example:

```
SELECT LAST_DAY('15-APR-1999') FROM DUAL -> 30-APR-99
```

LAST_VALUE(expression [IGNORE NULLS]) OVER (window_clause)

Returns the last value in an ordered set of values. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

SELECT c1, LAST_VALUE(c1) OVER (ORDER BY c1) FROM FIVE_NUMS

```
1    5
2    5
3    5
4    5
5    5
```

LEAD(expression [,offset][,default]) OVER(window_clause)

An analytic function that provides access to more than one row of a table at the same time without a self join. *LEAD* provides a "leading" value in the result set that is *offset* rows ahead of the current row. The *default* value is used for the last *offset* rows in the result set, since the "leading" value is undefined for these rows. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

SELECT c1, LEAD(c1, 2) OVER (ORDER BY c1) FROM FIVE_NUMS

```
1    3
2    4
3    5
4
5
```

LEAST(expression [,...n])

Returns the least of the list of expressions. For example:

SELECT LEAST(10,5,3,7) FROM DUAL -> 3

LENGTH(string)

Returns the integer length of *string*, or NULL if *string* is NULL. For example:

SELECT LENGTH('DUCK') FROM DUAL -> 4

LENGTHB(string)

Returns the length of *char* in bytes; otherwise, the same as *LENGTH*. For example:

SELECT LENGTHB('DUCK') FROM DUAL -> 4

Use *LENGTHB* for bytes, *LENGTHC* for UNICODE complete characters, *LENGTH2* for UNICODE UCS2 code points, and *LENGTH4* for UNICODE UCS4 code points.

LNNVL(condition)

Returns *true* if *condition* is false or if one of the operands in *condition* is NULL. For example:

SELECT COUNT(*) FROM authors WHERE LNNVL(contract <> 1) -> 4

LOCALTIMESTAMP [(precision)]

Returns a *TIMESTAMP* value for the current date and time. This function is similar to *CURRENT_TIMESTAMP*, with the exception that this function does not return a *TIME_ZONE* value with the *TIMESTAMP*. For example:

SELECT LOCALTIMESTAMP FROM DUAL -> '15-APR-05 03.15.00 PM'

LOG(base_number, number)

Returns the logarithm of any *base_number* of *number*. For example:

SELECT LOG(50,10) FROM DUAL -> .58859191

LPAD(string1, number [,string2])

Returns *string1*, left-padded to length *number* using characters in *string2*; *string2* defaults to a single blank. For example:

SELECT LPAD('ucks',5,'d') FROM DUAL -> 'ducks'

LTRIM(string[, set])

Removes all characters in *set* from the left of *string*. *Set* defaults to a single blank. For example:

SELECT LTRIM(' Howdy! ',' ') FROM DUAL -> 'Howdy! '

MAKE_REF({table_name | view_name} , key [,...n])

Creates a reference (*REF*) to a row of an object view or a row in an object table whose object identifier is primary key-based.

MEDIAN(expression) OVER (partitioning)

Returns the median value in an ordered set of numeric or datetime values. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *partitioning* clause. For example:

SELECT MEDIAN(c1) FROM FIVE_NUMS -> 3

MONTHS_BETWEEN(date1, date2)

Returns the number of months between dates *date1* and *date2*. When *date1* is later than *date2*, the result is positive. If it is earlier, the result is negative. For example:

SELECT MONTHS_BETWEEN('15-APR-2000', '15-JUL-1999') FROM DUAL -> 9

NANVL(a, b)

Returns *b* when *a* is not a number (*NaN*); returns *a* otherwise. Expression *a* must evaluate to a *BINARY_FLOAT* or *BINARY_DOUBLE* number, which are the only number types that permit storing *NaN*. For example:

SELECT c1, NANVL(c1, 0) FROM NUMS

1.0E+000 1.0E+000

2.0E+000 2.0E+000

Nan 0

NCHAR(number)

A synonym for *CHR(number) USING NCHAR_CS*.

NEW_TIME(date, time_zone1, time_zone2)

Returns the date and time in *time_zone2* using *date* as the input date/time, and using *time_zone1* as the originating time zone. For example:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH12:MI:SS'
```

```
SELECT NEW_TIME(TO_DATE('04-15-99 08:22:31', 'MM-DD-YY HH12:MI:SS'),  
'AST', 'PST') FROM DUAL
```

```
15-APR-2099 04:22:31
```

Time_zone1 and *Time_zone2* may be any of these text strings:

AST, ADT: Atlantic Standard or Daylight Time

BST, BDT: Bering Standard or Daylight Time

CST, CDT: Central Standard or Daylight Time

EST, EDT: Eastern Standard or Daylight Time

GMT: Greenwich Mean Time

HST, HDT: Alaska-Hawaii Standard or Daylight Time

MST, MDT: Mountain Standard or Daylight Time

NST: Newfoundland Standard Time

PST, PDT: Pacific Standard or Daylight Time

YST, YDT: Yukon Standard or Daylight Time

NEXT_DAY(date, string)

Returns the date of the first weekday named by *string* that is later than *date*. The argument *string* must be either the full name or the abbreviation of a day of the week in the date language of the session. For example:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'
```

```
SELECT NEXT_DAY('15-APR-1999', 'SUNDAY') FROM DUAL
```

```
18-APR-1999
```

NLS_CHARSET_DECL_LEN(bytecnt, csid)

Returns the declaration width (*bytecnt*) of an *NCHAR* column using the character set ID (*csid*) of the column.

NLS_CHARSET_ID(text)

Returns the NLS character set ID number corresponding to *text*.

NLS_CHARSET_NAME(number)

Returns the *VARCHAR2* name for the NLS character set corresponding to the ID *number*.

NLS_INITCAP(string [, nlsparameter])

Returns *string* with the first letter of each word in uppercase and all other letters in lowercase. The *nlsparameter* offers special linguistic sorting features.

NLS_LOWER(string [, nlsparameter])

Returns *string* with all letters lowercase. The *nlsparameter* offers special linguistic sorting features.

NLSSORT(*string* [,*nlsparameter*])

Returns the string of bytes used to sort *string*. The *nlsparameter* offers special linguistic sorting features.

NLS_UPPER(*string* [,*nlsparameter*])

Returns *string* with all letters uppercase. The *nlsparameter* offers special linguistic sorting features.

NTILE(*expression*) *OVER* ([*partitioning*] *ordering*)

Divides an ordered data set into a number of groups numbered 1 to *expression* and assigns the appropriate group number to each row. Rows are allocated to each group so that the number of rows per group varies by no more than one. See [Section 4.3](#) earlier in this chapter for details on the *partitioning* and *ordering* clauses. For example:

SELECT c1, NTILE(4) OVER (ORDER BY c1) FROM FIVE_NUMS

```
1  1
2  1
3  2
4  3
5  4
```

NULLIF(*expr1*, *expr2*)

Returns NULL if *expr1* is equal to *expr2*. If the two expressions are not equal, then the function returns *expr1*. A NULL value, if passed as one of the expressions, can only be specified as the second expression *expr2*. For example:

SELECT c1, c2, NULLIF(c1, c2) FROM NUMS

```
1  2  1
2  2
3  2  3
```

NUMTODSINTERVAL(*number*, *string*)

Converts *number* to an *INTERVAL DAY TO SECOND* literal, where *number* is a number or an expression resolving to a number, such as a numeric datatype column. The second argument, *string*, can be 'DAY', 'HOUR', 'MINUTE', or 'SECOND', which specifies how to interpret *number*. For example:

SELECT NUMTODSINTERVAL(100, 'DAY') FROM DUAL

```
+000000100 00:00:00.000000000
```

NUMTOYMINTERVAL(*number*, *string*)

Converts *number* to an *INTERVAL YEAR TO MONTH* literal, where *number* is a number or an expression resolving to a number, such as a numeric datatype column. The second argument, *string*, can be 'YEAR' or 'MONTH', which specifies how to interpret *number*. For example:

SELECT NUMTOYMINTERVAL(100, 'YEAR') FROM DUAL

```
+000000100-00
```

NVL(*expression1*, *expression2*)

If *expression1* is NULL, *expression2* is returned in place of that NULL value. Otherwise, *expression1* is returned. *expression1* and *expression2* may be any datatype. For example:

```
SELECT NVL(2,29) FROM DUAL -> 2
```

NVL2(expression1, expression2, expression3)

Similar to *NVL*, except that if *expression1* is not NULL, *expression2* is returned. If *expression1* is NULL, *expression3* is returned. The expressions may be any datatype except *LONG*. For example:

```
SELECT NVL2(1,3,5) FROM DUAL -> 3
```

ORA_HASH(expression [, buckets [, seed]])

Computes a hash value from *expression* and returns a bucket number based on the computed hash value. The optional *buckets* argument is the maximum bucket number to use, which is one less than the total number of buckets, since the bucket numbering starts at 0. The default for *buckets* is 4,294,967,295. The optional *seed* value is used to seed the hashing function so that multiple results can be produced from the same data by only changing the *seed* value. The *seed* value defaults to 0. This example psuedo-randomly assigns all numbers to one of two buckets and returns those assigned to the first bucket, which will be a sample of roughly one half of the values:

```
SELECT C1 FROM FIVE_NUMS WHERE
```

```
ORA_HASH( C1, 1, TO_CHAR( SYSTIMESTAMP, 'SSSS.FF' ) ) = 0
```

```
1
```

```
5
```

PATH(number)

Returns the path specified by the *UNDER_PATH* condition with the correlation variable *number* in an XML query. See the Oracle SQL Reference for more information.

POWERMULTISET(nested_table) POWERMULTISET_BY_CARDINALITY(nested_table, cardinality)

Returns a nested table of nested tables of all nonempty subsets of the input nested table in the *nested_table* parameter. *POWERMULTISET_BY_CARDINALITY* has an additional parameter that can be used to limit the subsets returned to a specified minimum cardinality. For more information, please see the Oracle SQL Reference.

PRESENTNNV(cell_reference, expr1, expr2)

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns *expr1* when *cell_reference* exists and is not NULL; otherwise, it returns *expr2*.

PRESENTV(cell_reference, expr1, expr2)

Relevant only in the interrow calculations performed within the *MODEL* clause of a *SELECT* statement. This function returns *expr1* when *cell_reference* exists; otherwise, it returns *expr2*.

PREVIOUS(cell_reference)

Relevant only in the inter-row calculations performed within the *ITERATE...[UNTIL]* section of a *SELECT*'s *MODEL* clause. This function returns the value held by *cell_reference* at the beginning of the iteration.

RATIO_TO_REPORT (value_exprs) OVER (partitioning)

Computes the ratio of a value in *value_exprs* to the sum of all *value_exprs* with each partition. If *values_expr* is NULL, the ratio-to-report value also is NULL. See [Section 4.3](#) earlier in this chapter for details on the *partitioning* clause. For example:

```
SELECT c1, RATIO_TO_REPORT(c1) OVER ( ) FROM FIVE_NUMS
```

```
1      .066666667
2      .133333333
3      .2
4      .266666667
5      .333333333
```

RAWTOHEX(raw)

Converts a *raw* value to a string (character datatype) of its hexadecimal equivalent. For example:

```
SELECT RAWTOHEX('Hi') FROM DUAL -> 4869
```

RAWTONHEX(raw)

Converts a *raw* value to an *NVARCHAR2* (character datatype) of its hexadecimal equivalent.

REF(table_alias)

REF takes a table alias associated with a row from an object table or an object view. A special reference value is returned for the object instance that is bound to the variable or row.

REFTOHEX(expression)

Converts *expression* to a character value containing its hexadecimal equivalent.

REGEXP_INSTR(string, pattern [, start_at [, occurrence [, roption [, mparam]]]])

Returns the character position within *string* matching the regular expression pattern in *pattern*. *REGEXP_INSTR* searches *string* with a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *roption* parameter can be 0 or 1 and specifies if the position returned is the first character matching the pattern or the character after. The default for *roption* is 0, which returns the position of the first character matching the pattern. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

'i'

matching is case-insensitive

'c'

matching is case-sensitive

'n'

the "." character matches newline characters

'm'

treat input string as multiple lines; use '^' to match the beginning of a line and '\$' to match the end of a line

For example:

```
SELECT REGEXP_INSTR( 'Hello, World!', '([ ^ ]*)!', 1, 1) FROM DUAL -> 8
```


REGEXP_REPLACE(string, pattern [, newstr [, start_at [, occurrence [, mparam]]]])

Returns the result of replacing all occurrences of the regular expression in *pattern* within *string* with another string *newstr*. *REGEXP_REPLACE* searches *string* with a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

'i'

matching is case-insensitive

'c'

matching is case-sensitive

'n'

the "." character matches newline characters

'm'

treat input string as multiple lines; use '^' to match the beginning of a line and '\$' to match the end of a line

For example:

```
SELECT REGEXP_REPLACE( 'Hello, World!', '([^\ ]*!)', 'Reader!') FROM DUAL
```

```
'Hello, Reader!'
```

REGEXP_SUBSTR(string, pattern [, start_at [, occurrence [, mparam]]])

Returns the substring within *string* matching the regular expression pattern *pattern*. *REGEXP_SUBSTR* searches *string* with a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

'i'

matching is case-insensitive

'c'

matching is case-sensitive

'n'

the `.` character matches newline characters

'm'

treat input string as multiple lines; use '^' to match the beginning of a line and '\$' to match the end of a line

For example:

```
SELECT REGEXP_SUBSTR( 'Hello, World!', '([^\ ]*!)' ) FROM DUAL
```

```
'World!'
```

REMAINDER(m, n)

Returns the remainder of *m* divided by *n*. This return value is equivalent to the expression:

$m-n*\text{ROUND}(m/n)$

The function *MOD* uses *FLOOR* instead of *ROUND*. For example:

SELECT REMAINDER(11, 4), MOD(11, 4) FROM DUAL

-1 3

REPLACE(*string*, *search_string* [,*replacement_string*])

Returns *string* with every occurrence of *search_string* replaced with *replacement_string*. For example:

SELECT REPLACE('change', 'e', 'ing') FROM DUAL -> 'changing'

ROUND (*number* [, *decimal*])

Returns *number* rounded to *decimal* places right of the decimal point. When *decimal* is omitted, *number* is rounded to an integer. Note that *decimal*, an integer, can be negative to round off digits left of the decimal point. For example:

SELECT ROUND(12345.6789, 2) FROM DUAL -> 12345.68

ROUND (*date* [, *format*])

Returns the *date* rounded to the unit specified by the *format* model. When *format* is omitted, date is rounded to the nearest day. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

SELECT ROUND(TO_DATE('15-APR-1999'), 'MONTH') FROM DUAL

01-APR-1999

ROWIDTOCHAR(*rowid*), *ROWIDTONCHAR*(*rowid*)

ROWIDTOCHAR converts a *rowid* value to the *VARCHAR2* datatype, 18 characters long. *ROWIDTONCHAR* converts *rowid* to an 18-character long *NVARCHAR2* value. For example:

SELECT ROWIDTOCHAR(ROWID) FROM NUMS

ABAsxDAAKAAAEEqAAA

ABAsxDAAKAAAEEqAAB

ABAsxDAAKAAAEEqAAC

ABAsxDAAKAAAEEqAAD

RPAD(*string1*, *number* [, *string2*])

Returns *string1*, right-padded to length *number* with the value of *string2*, repeated as needed. *string2* defaults to a single blank. For example:

SELECT RPAD('duck',8,'s') FROM DUAL -> 'duckssss'

RTRIM(*string* [,*set*])

Returns *string*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. For example:

SELECT RTRIM(' welcome ', ' ') FROM DUAL -> ' welcome'

SCN_TO_TIMESTAMP(scn)

Returns the timestamp associated with the System Change Number (SCN) argument. For example:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM NUMS WHERE c1 = 1
```

```
15-APR-04 02.56.05.000000000 PM
```

SESSIONTIMEZONE

Returns the session's time zone offset. For example:

```
SELECT SESSIONTIMEZONE FROM DUAL -> -06:00
```

SET(nested_table)

Returns a nested table of distinct elements from the input nested table. For more information, please see the Oracle SQL Reference.

SIGN(number)

When *number* < 0, returns -1. When *number* = 0, returns 0. When *number* > 0, returns 1. For example:

```
SELECT SIGN(-3.1415926) FROM DUAL -> -1
```

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

SINH(number)

Returns the hyperbolic sine of *number*. For example:

```
SELECT SINH(180) FROM DUAL -> 7.4469E+77
```

SOUNDEX(string)

Returns a character string containing the phonetic representation of *string*. This function allows words that are spelled differently but sound alike in English to be compared for equality. For example:

```
SELECT SOUNDEX('thimble') FROM DUAL -> 'T514'
```

STATS_BINOMIAL_TEST, STATS_CROSSTAB, STATS_F_TEST, STATS_KS_TEST, STATS_MODE, STATS_MW_TEST, STATS_ONE_WAY_ANOVA, STATS_T_TEST_ONE, STATS_T_TEST_PAired, STATS_T_TEST_INDEP, STATS_T_TEST_INDEPU, STATS_WSR_TEST

Oracle provides many sophisticated statistical functions. For further information on the *STATS_** functions, please see the Oracle SQL Reference.

STDDEV([DISTINCT | ALL] expression) [OVER (window_clause)]

Returns a sample standard deviation of a set of numbers shown as *expression*. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV(col1) FROM NUMS -> 5.71547607
```

STDDEV_SAMP(*expression*) [*OVER* (*window_clause*)]

Computes the cumulative sample standard deviation and returns the square root of the sample variance. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV_SAMP(col1) FROM NUMS    -> 5.71547607
```

STDEV_POP(*expression*) [*OVER* (*window_clause*)]

Computes the population standard deviation and returns the square root of the population variance. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV_POP(col1) FROM NUMS    -> 4.94974747
```

SUBSTR(*string*, *start* [*FROM starting_position*] [*FOR length*])

Refer to the earlier section on *SUBSTR*. For example:

```
SELECT SUBSTR('Hello, World!',8,10) FROM DUAL -> 'World!'
```

SUBSTRB(*extraction_string* [, *length*])

Returns the portion of *string* beginning at *start* and continuing for *length* characters. If *length* is omitted, all characters from *start* onward are returned. If *start* is negative, it represents an offset from the right edge of the *string*. For example:

```
SELECT SUBSTR ('Hello, World!', 8)
```

```
FROM DUAL -> World!
```

Use *SUBSTRB* for bytes, *SUBSTRC* for UNICODE complete characters, *SUBSTR2* for UNICODE UCS2 code points, and *SUBSTR4* for UNICODE UCS4 code points.

SYS_CONNECT_BY_PATH(*column*, *char*)

For hierarchical queries, *SYS_CONNECT_BY_PATH* returns the path from the root to the node with the column name specified by the *column* parameter. The *char* parameter specifies the node separator for the return path. For more on Oracle hierarchical queries, please refer to the Oracle SQL Reference.

SYS_CONTEXT(*namespace*, *attribute* [, *length*])

Returns the value of *attribute* associated with the context *namespace*, usable in both SQL and PL/SQL statements. For example:

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL
```

```
'LOGIN'
```

SYS_DBURIGEN(*column* [, *rowid*] [, ...] [, 'text() '])

Returns a URL that can be used as a unique reference to the row specified by the *column* parameter. For columns that don't hold unique values, a *rowid* can be used directly after the column it identifies to guarantee that the URL only points to one row. Use the 'text()' option if you want the URL to point to the text within an XML document, instead of the document itself.

SYS_EXTRACT_UTC(*datetime*)

Returns the *datetime* argument converted to a UTC datetime value. For example:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2004-04-15 11:59:59.00 -08:00')  
FROM DUAL  
  
'15-APR-04 07.59.59.000000000 PM'
```

SYS_GUID()

Generates and returns a globally unique identifier (*RAW* value) made up of 16 bytes. For example:

```
SELECT SYS_GUID( ) FROM DUAL -> C0FD3FDC30148EAE030440A49096A41
```

SYS_TYPEID(object_value)

Returns the type ID of the *object_value* parameter.

SYS_XMLAGG(expr [, format])

Returns a single XML document created by aggregating the XML documents or fragments in the *expr* parameter. The optional *format* parameter can be used to format the XML document. For more information, refer to the Oracle SQL Reference.

SYS_XMLGEN(expression [, format])

Returns a single XML document created from the *expression*. The optional *format* parameter can be used to format the XML document. For more information, refer to the Oracle SQL Reference.

SYSDATE

Returns the current date and time on the system in which the database is hosted. The value returned is of type *DATE*. For example:

```
SELECT SYSDATE FROM DUAL -> 26-JUN-2003
```

SYSTIMESTAMP

Returns the current date and time on the system in which the database is hosted. The value returned is of type *TIMESTAMP*. For example:

```
SELECT SYSTIMESTAMP FROM DUAL  
  
26-JUN-2003 11.15.00.000000 PM -06:00
```

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) FROM DUAL -> -0.000093
```

TANH(number)

Returns the hyperbolic tangent of *number*. For example:

```
SELECT TANH(180) FROM DUAL -> 1
```

TIMESTAMP_TO_SCN(timestamp_value)

Returns the approximate system change number (*SCN*) associated with that timestamp. The return value is of

type *NUMBER*.

TO_BINARY_DOUBLE (*expr* [, *format* [, *nls_parameter*]])

Converts *expr* to a *BINARY_DOUBLE* in the format specified by the *format* parameter. If *expr* is a character expression, the *format* and *nls_parameter* options have equivalent meanings, as they do in the *TO_CHAR* function. If *expr* is a numeric expression, then *format* and *nls_parameter* must be omitted. For example:

SELECT c1, TO_BINARY_DOUBLE(c1) FROM NUMS

1	1.0E+000
2	2.0E+000
3	3.0E+000

TO_BINARY_FLOAT(*expr* [, *format* [, *nls_parameter*]])

Converts *expr* to a *BINARY_FLOAT* in the format specified by the *format* parameter. If *expr* is a character expression, the *format* and *nls_parameter* options have equivalent meanings as they do in the *TO_CHAR* function. If *expr* is a numeric expression, then *format* and *nls_parameter* must be omitted. For example:

SELECT c1, TO_BINARY_FLOAT(c1) FROM NUMS

1	1.0E+000
2	2.0E+000
3	3.0E+000

TO_CHAR (*character_expr*)

Converts *character_expr* to the database character set. For example:

SELECT TO_CHAR('Howdy') FROM DUAL

Howdy

TO_CHAR (*date* | *interval* [, *format* [, *nls_parameter*]])

Converts *date* or *interval* to a *VARCHAR2* in the format specified by the date *format*. When *format* is omitted, *date* is converted to the default date format. The *nls_parameter* option offers additional control over formatting. Listed below are the available specifiers for *format* and their meanings.

AD or *A.D.*

AD indicator

AM or *A.M.*

Meridian indicator

BC or *B.C.*

BC indicator

D

Day of week (1-7)

DAY

Name of day

DD

Day of month (1-31)

DDD

Day of year (1-366)

DL

Long date format

DS

Short date format

DY

Abbreviated name of day

FF

Fractional seconds; to specify the precision, include a number (1-9) after the FF specifier

HH or HH12

Hour of day (1-12)

HH24

Hour of day (0-23)

J

Julian day; the number of days since January 1, 4713 BC

MI

Minute (0-59)

MM

Month (01-12)

MON

Abbreviated name of month

RM

Roman numeral month (I-XII)

SS

Second (0-59)

SSSSS

Seconds past midnight (0-86,399)

SYYY

Four-digit year; BC dates are prefixed with a minus sign

TS

Short time format.

TZD

Daylight savings information (example: PST versus PDT)

TZH

Time zone hour

TZM

Time zone minute

TZR

Time zone region

X

Local radix character

Y, YY, or YYY

One, two, or three digits for the year

Y,YYY

Year with comma

YYYY

Four-digit year

For example:

```
SELECT TO_CHAR(TO_DATE('15-APR-1999'), 'MON-DD-YYYY') FROM DUAL
```

```
APR-15-1999
```

TO_CHAR (number [, format [, nls_parameter]])

Converts *number* to a *VARCHAR2* in the *format* specified. When *format* is omitted, *number* is converted to a string long enough to hold the number. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_CHAR(123.45, '$999.99') FROM DUAL -> $123.45
```


TO_CLOB (expr)

Converts the character expression given by *expr* to the *CLOB* datatype. For example:

```
SELECT LENGTH(TO_CLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_DATE(string [, format [, nls_parameter]])

Converts *string* (in *CHAR* or *VARCHAR2*) to a *DATE* datatype. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_DATE('15/04/1999', 'DD/MM/YYYY') FROM DUAL  
  
15-APR-1999
```

TO_DSINTERVAL(string [, nls_parameter])

Converts the character expression given by *string* to the *INTERVAL DAY TO SECOND* datatype. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT CURRENT_DATE, CURRENT_DATE-TO_DSINTERVAL('14 00:00:00') FROM DUAL  
  
15-APR-2003    01-APR-2003
```

TO_LOB(long_column)

Usable only by *LONG* or *LONG RAW* expressions, it converts *LONG* or *LONG RAW* values in the column *long_column* to *LOB* values. It is usable only in the *SELECT* list of a subquery in an *INSERT* statement.

TO_MULTI_BYTE(string)

Returns *string* with all of its single-byte characters converted to their corresponding multibyte characters.

TO_NCHAR(expr [, format [, nls_parameter]])

Synonymous with the *TO_CHAR* function, except the return datatype is *NCHAR*. For more on valid format specifiers, see the *TO_CHAR* function.

TO_NCLOB (expr)

Converts the character expression given by *expr* to the *NCLOB* datatype. For example:

```
SELECT LENGTH(TO_NCLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_NUMBER(string [, format [, nls_parameter]])

Converts a numeric *string* (of *CHAR* or *VARCHAR2* datatype) to a value of *NUMBER* datatype, optionally in the format specified by the *format* model. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_NUMBER('12345') FROM DUAL    -> 12345
```

TO_SINGLE_BYTE(string)

Returns *string* with all of its multibyte characters converted to their corresponding single-byte characters.

TO_TIMESTAMP(string [, format [, nls_parameter]])

Converts the character expression provided by *string* to the *TIMESTAMP* datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TO_TIMESTAMP(CURRENT_DATE) FROM DUAL -> 04-MAY-04 12.00.00 AM
```

TO_TIMESTAMP_TZ(*string* [, *format* [, *nls_parameter*]])

Converts the character expression provided by *string* to the *TIMESTAMP WITH TIME ZONE* datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TO_TIMESTAMP_TZ('15-04-2006', 'DD-MM-YYYY') FROM DUAL
```

```
15-APR-06 12.00.00.000000000 AM -07:00
```

TO_YMINTERVAL(*string*)

Converts the character expression provided by *string* to the *INTERVAL YEAR TO MONTH* datatype. For example:

```
SELECT TO_DATE('29-FEB-2000')+TO_YMINTERVAL('04-00') FROM DUAL
```

```
29-FEB-04
```

TRANSLATE(*char_value*, *from_text*, *to_text*)

Returns *char_value* with each occurrence of a character in *from_text* replaced by its corresponding character in *to_text*. For example:

```
SELECT TRANSLATE('foobar', 'fa', 'bu') FROM DUAL -> 'boobur'
```

TRANSLATE (*text* USING [*CHAR_CS* | *NCHAR_CS*])

Converts *text* into the character set specified. Use *CHAR_CS* to convert *text* to the *CHAR* datatype or *NCHAR_CS* to convert *text* to the *NCHAR* datatype. For example:

```
SELECT TRANSLATE(N'foobar' USING CHAR_CS) FROM DUAL
```

```
'foobar'
```

TREAT (*expr* AS [*REF*] [*schema.*]*type*)

Converts *expr* from its declared type to the type specified in the *type* parameter. For more information on the usage of this function, please look to the SQL Reference for the Oracle Database.

TRUNC (*base* [, *number*])

Returns *base* truncated to *number* decimal places. When *number* is omitted, *base* is truncated to an integer. *number* can be negative to truncate (make zero) digits left of the decimal point. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TRUNC('123.456', 2) FROM DUAL -> 123.45
```

TRUNC (*date* [, *format*])

Returns *date* truncated to the unit specified by *format*. When *format* is omitted, *date* is truncated to the nearest whole day. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TRUNC(TO_DATE('15/04/1999', 'MM/DD/YYYY'), 'YYYY')
```

FROM DUAL

1999

See *TO_CHAR* for a list of format specifiers.

TZ_OFFSET ({ *expr* | *SESSIONTIMEZONE* | *DBTIMEZONE* }

Returns the time zone offset corresponding to the argument. The character expression *expr* can either be the name of the time zone or a time zone offset. The *SESSIONTIMEZONE* and *DBTIMEZONE* arguments provide the time zone for the session or database, respectively. For example:

SELECT TZ_OFFSET('+08:00'), TZ_OFFSET(SESSIONTIMEZONE) FROM DUAL

+08:00 -07:00

UID

Returns an integer that uniquely identifies the currently logged on session user. No parameters are needed. For example:

SELECT UID FROM DUAL -> 47

UNISTR(*string*)

Converts *string* to the *NCHAR* datatype while converting any UNICODE encoding values within *string*. For example:

SELECT UNISTR('Ei Ni\00F1o') FROM DUAL -> 'El Niño'

UPDATEXML(*instance*, *xpath*, *expr* [, *namespace*])

Updates the values held by nodes within *instance* to the new value in *expr*. Only those nodes returned by the XPath query contained in the *xpath* parameter are updated. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

SELECT UPDATEXML(XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),

'/foo/bar', '<bar>Bye, World!</bar>') from DUAL

Bye, World!

USERENV(*option*)

Returns information about the current session in *VARCHAR2*. This function has been deprecated and is only provided for backward compatibility. *USERENV* is a synonym for *SYS_CONTEXT*('USER_ENV', *option*). Look to the *SYS_CONTEXT* function with the *USERENV* namespace for current functionality. For example:

SELECT USERENV('LANGUAGE') "Language" FROM DUAL

'AMERICAN_AMERICA.AL32UTF8'

VALUE(*table_alias*)

Takes a *table_alias* associated with a row in an object table and returns the object instance stored within the object table for that row.

VARIANCE([*DISTINCT*] *expression*) [*OVER* (*window_clause*)]

Returns the variance of *expression* calculated as follows: 0 if the number of rows in *expression* = 1; *VAR_SAMP* if the number of rows in *expression* > 1. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For

example:

```
SELECT VARIANCE(col1) FROM NUMS -> 32.6666667
```

VAR_POP(*expression*) [*OVER* (*window_clause*)]

Returns the population variance of the set of numbers represented by *expression* after discarding the NULLs in the set. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT VAR_POP(col1) FROM NUMS -> 24.5
```

VAR_SAMP(*expression*) [*OVER* (*window_clause*)]

Returns the sample variance of the set of numbers represented by *expression* after discarding the NULLs in the set. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT VAR_SAMP(col1) FROM NUMS -> 32.6666667
```

VSIZE(*expression*)

Returns the number of bytes in the internal representation of *expression*. When *expression* is *NULL*, returns *NULL*. For example:

```
SELECT vsize(1) FROM DUAL -> 2
```

XMLAGG(*instance* [, *order_by*])

An aggregate function that returns an XML document from a table of XML fragments contained in *instance*. The optional *order_by* clause allows the XML fragments in the result to be ordered. For more information on XML queries refer to the Oracle SQL Reference.

XMLCOLATTVAL(*expr* [*AS alias*] [,...])

Returns an XML fragment from the *expr* arguments. The optional *AS* clause can be used to change the value of the name attribute. For more information on XML queries, refer to the Oracle SQL Reference.

XMLCONCAT(*instance* [,...])

Returns an XML instance that is the union of all XML *instance* parameters. For more information on XML queries refer to the Oracle SQL Reference.

XMLELEMENT([*NAME*] *name* [, **XMLATTRIBUTES**(*expr* [*AS alias*] [,...])] [, *value* [,...]])

Returns an **XMLELEMENT** with the name specified in the *name* parameter and attributes specified in the optional **XMLATTRIBUTES** clause. The *value* parameters provide the values of the **XMLELEMENT** result. For more information on XML queries refer to the Oracle SQL Reference.

XMLFOREST(*value* [*AS alias*] [,...])

Returns an XML fragment constructed from the values provided in the *value* parameters. The optional *AS* clause can be used to change the enclosing tag name. For more information on XML queries refer to the Oracle SQL Reference.

XMLSEQUENCE(*instance*)

Returns an array of XML fragments constructed from the top-level nodes in the XML instance provided by the *instance* parameter. For more information on XML queries, refer to the Oracle SQL Reference.

XMLTRANSFORM(*instance*, *stylesheet*)

Returns the result of applying the XSL stylesheet in the *stylesheet* parameter to the XML document contained in *instance*. For more information on XML queries, refer to the Oracle SQL Reference.

4.5.4 PostgreSQL-Supported Functions

This section lists the functions specific to PostgreSQL, with examples and corresponding results.

ABSTIME(timestamp)

Converts the *timestamp* value to *ABSTIME* type. This function is provided for backwards compatibility and may be removed in future versions. For example:

```
SELECT ABSTIME(CURRENT_TIMESTAMP) -> 2003-06-24 00:19:17-07
```

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS(0) -> 1.570796
```

AGE(timestamp)

Same meaning as *AGE(CURRENT_DATE, timestamp)*.

AGE(timestamp, timestamp)

Returns the time between the two *timestamp* values. For example:

```
SELECT AGE('2003-12-31', CURRENT_TIMESTAMP)
```

```
6 mons 7 days 00:34:41.658325
```

AREA(object)

Returns the area of an item. For example:

```
SELECT AREA(BOX '((0,0),(1,1))) -> 1
```

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN(0) -> 0.000000
```

ATAN(number)

Returns the arctangent of *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN(3.1415) -> 1.262619
```

ATAN2(float1, float2)

Returns the arctangent of the two *float* values. *ATAN2(x,y)* is similar to *ATAN(y/x)*, with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

SELECT ATAN2(3.1415926, 0) -> 1.5707963267949

BOX(box, box)

Returns a *BOX* created by the intersection of the two boxes. If the two boxes do not intersect, the return value is NULL. For example:

SELECT BOX(BOX '((-1,-1),(1,1))', BOX '((0,0),(1,1))')

(1,1),(0,0)

BOX(circle)

Returns a *BOX* with vertices that intersect *circle* so that the box is the maximum size that can be fully contained within *circle*. For example:

SELECT BOX(CIRCLE '((0,0),2.0)')

(1.41421356237309,1.41421356237309),

(-1.41421356237309,-1.41421356237309)

BOX(point, point)

Returns a *BOX* with the two point arguments as opposite corners. For example:

SELECT BOX(POINT(0,0), POINT(1,1)) -> (1,1),(0,0)

BOX(polygon)

Converts *polygon* to a *BOX*. For example:

SELECT BOX(POLYGON '((0,0),(1,1),(2,0))') -> (2,1),(0,0)

BROADCAST(inet)

Constructs a broadcast address as text. For example:

SELECT BROADCAST('192.168.1.5/24') -> '192.168.1.255/24'

CBRT(float8)

Returns the cube root of *float8*. For example:

SELECT CBRT(8) -> 2

CENTER(object)

Returns a *POINT* object to the center of the argument. For example:

SELECT CENTER(CIRCLE '((0,0), 2.0)') -> (0,0)

CHAR(text)

Converts *text* to the *CHAR* type.

CHAR_LENGTH(string) or *CHARACTER_LENGTH(string)*

Returns the length of *string* in characters.

CIRCLE(box)

Returns a *CIRCLE* contained within *box*. For example:

SELECT CIRCLE(BOX '((0,0),(1,1)')) -> <(0.5,0.5),0.707106781186548>

CIRCLE(point, float8)

Converts *point* to a *CIRCLE* with *float8* for the radius. For example:

SELECT CIRCLE(POINT '(0,0)', 2.0) -> <(0,0),2>

COALESCE(list)

Returns the first non-NULL value in *list*. For example:

SELECT COALESCE(NULL,1,2,3,NULL) -> 1

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

COT(number)

Returns the cotangent of *number*. For example:

SELECT COT(3.1415) -> -10792.88993953

DATE_PART(text, value)

Equivalent to *EXTRACT(text, value)*; for more details on the usage of *EXTRACT*, see the section about *EXTRACT* in [Section 4.4](#).

DATE_TRUNC(precision, timestamp)

Truncates *timestamp* to the specified *precision*. For example:

SELECT DATE_TRUNC('hour', TIMESTAMP '2003-04-15 23:58:30')

2003-04-15 23:00:00

DEGREES(float8)

Converts radians to degrees. For example:

SELECT DEGREES(3.1415926) -> 179.999996929531

DIAMETER(circle)

Returns the diameter of *circle*. For example:

SELECT DIAMETER(CIRCLE(POINT '(0,0)', 2.0)) -> 4

FLOAT(int)

Converts *int* to a floating point.

FLOAT4(int)

Converts *int* to a floating point.

HEIGHT(box)

Returns the vertical size of *box*. For example:

SELECT HEIGHT(BOX '((0,0),(1,1))) -> 1

HOST(inet)

Extracts the host address as text. For example:

SELECT HOST('192.168.1.5/24') -> '192.168.1.5'

INITCAP(text)

Converts the first letter of each word to uppercase. For example:

SELECT INITCAP('my name is inigo montoya.')

'My Name Is Inigo Montoya.'

INTEGER(float)

Converts a floating point to integer.

INTERVAL(reltime)

Converts *reltime* to an *INTERVAL*.

ISCLOSED(path)

Returns 't' if *path* is closed, 'f' if open. For example:

SELECT ISCLOSED(PATH '((0,0),(1,1),(2,0))) -> 't'

SELECT ISCLOSED(PATH '[(0,0),(1,1),(2,0)]') -> 'f'

ISFINITE(interval)

Returns 'f' if *interval* is open, 't' otherwise. For example:

SELECT ISFINITE(INTERVAL '4 hours') -> 't'

ISFINITE(timestamp)

Returns 'f' if *timestamp* is either invalid or infinite, 't' otherwise. For example:

```
SELECT ISFINITE(TIMESTAMP '2001-02-16 21:28:30') -> 't'
```

ISOPEN(path)

Returns an open *path*. For example:

```
SELECT ISOPEN(PATH '((0,0),(1,1),(2,0)))' -> 'f'
```

```
SELECT ISOPEN(PATH '[(0,0),(1,1),(2,0)]') -> 't'
```

LENGTH(object)

Returns the length of *object*. For example:

```
SELECT LENGTH('Howdy!') -> 6
```

```
SELECT LENGTH(PATH '((-1,0),(1,0))') -> 4
```

LOG(float8)

Returns a base-10 logarithm. For example:

```
SELECT LOG( 100 ) -> 2
```

LPAD(text,int,text)

Returns a left-pad string to the specified length. For example:

```
SELECT LPAD('Duck', 10, 's') -> 'sssssDuck'
```

LSEG(box)

Converts a *box* diagonal to a line segment. For example:

```
SELECT LSEG(BOX '((-1,0),(1,0))') -> [(1,0),(-1,0)]
```

LSEG(point,point)

Converts *points* to a line segment. For example:

```
SELECT LSEG(POINT '(-1,0)', POINT '(1,0)') -> [(-1,0),(1,0)]
```

LTRIM(text)

Returns *text* with all leading whitespace removed. For example:

```
SELECT LTRIM(' Howdy! ') -> 'Howdy! '
```

MASKLEN(cidr)

Returns the netmask length in *cidr*. For example:

```
SELECT MASKLEN('192.168.1.5/24') -> 24
```

NETMASK(inet)

Returns the netmask for *inet*. For example:

```
SELECT NETMASK('192.168.1.5/24') -> '255.255.255.0'
```

NETWORK(inet)

Returns the network part of *inet*. For example:

```
SELECT NETWORK('192.168.1.5/24') -> '192.168.1.0/24'
```

NPOINTS(object)

Returns the number of points in *object*. For example:

```
SELECT NPOINTS(POLYGON '((1,1),(0,0)')) -> 2
```

NULLIF(input, value)

Returns NULL if *input* = *value*, else returns *input*. For example:

```
SELECT NULLIF( 5, 6 ), NULLIF( 5, 5 )
```

```
5 NULL
```

PATH(polygon)

Converts *polygon* to a path. For example:

```
SELECT PATH( '((0,0),(1,1),(2,0))' )
```

```
((0,0),(1,1),(2,0))
```

PCLOSE(path)

Converts *path* to a closed path. For example:

```
SELECT PCLOSE(PATH '['(0,0),(1,1),(2,0)'])
```

```
((0,0),(1,1),(2,0))
```

PI()

Returns the constant pi.

POLYGON(path)

Converts *path* to a *POLYGON*. For example:

```
SELECT POLYGON(PATH '['(0,0),(1,1),(2,0)'])
```

```
((0,0),(1,1),(2,0))
```

POINT(circle)

Returns the center of *circle*. For example:

SELECT POINT(CIRCLE '((0,0), 2.0)') -> (0,0)

POINT(lseg1,lseg2)

Returns the intersection of two line segments. For example:

SELECT POINT(LSEG '((-1,0),(1,0))', LSEG '((-2,-2),(2,2))')
(0,0)

POINT(polygon)

Returns the center point of *polygon*. For example:

SELECT POINT(POLYGON '((0,0),(1,1),(2,0))')
(1,0.3333333333333333)

POLYGON(box)

Returns a 4-point polygon. For example:

SELECT POLYGON(BOX '((0,0),(1,1))')
((0,0),(0,1),(1,1),(1,0))

POLYGON(circle)

Synonym for *POLYGON(12, circle)*.

POLYGON(npts,circle)

Returns and approximation of *circle* as a polygon with *npts* vertices. For example:

SELECT POLYGON(6, CIRCLE '((0,0),2.0)')
((-2,0),
(-0.999999999994107,1.73205080757228),
(1.0000000001179,1.73205080756207),
(2,-2.04136478690279e-11),
(0.99999999976428,-1.73205080758249),
(-1.0000000002946,-1.73205080755187))

POPEN(path)

Converts *path* to an open path. For example:

SELECT POPEN(PATH '((0,0),(1,1),(2,0))')
[(0,0),(1,1),(2,0)]

POW (number,exponent)

Raises a *number* to the specified *exponent*. For example:

SELECT POW(2, 3) -> 8

RADIANS(float8)

Converts degrees to radians. For example:

SELECT RADIANS(180) -> 3.14159265358979

RADIUS(circle)

Returns the radius of *circle*. For example:

SELECT RADIUS(CIRCLE '((0,0), 2.0)') -> (0,0)

RELTIME(interval)

Converts *interval* to a *RELTIME*. Provided for backward compatibility and may be removed in a future release.

ROUND(number)

Rounds *number* to the nearest integer. For example:

SELECT ROUND(5.5) -> 6

RPAD(text,length,char)

Pads *text* to the specified *length* using *char*. For example:

SELECT RPAD('Duck', 10, 's') -> 'Duckssssss'

RTRIM(text)

Returns *text* with all trailing whitespace removed. For example:

SELECT RTRIM(' St. Lucia ') -> ' St. Lucia'

SET_MASKLEN(inet, size)

Sets the netmask length for *inet* to *size*. For example:

SELECT SET_MASKLEN('192.168.1.5/24',16)

'192.168.1.5/16'

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SUBSTRING(string [FROM start] [FOR bytes]), SUBSTR(string,start[,bytes])

Extracts a substring of length *bytes* from *string* starting at the character position *start*. If *bytes* is omitted, then the length returned is the remainder of the *string* from the *start* position. For example:

SELECT SUBSTRING('Inigo Montoya' FROM 7 FOR 4) -> 'Mont'

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

SELECT TAN(3.1415) -> -0.000093

TEXT(char)

Converts *char* to *TEXT* type.

TIMESTAMP(date [,time])

Converts *date* to a timestamp.

TO_CHAR(expression, text)

Converts *expression* to a string. For example:

SELECT TO_CHAR(NUMERIC '-125.8', '999D99S') -> 125.80-

SELECT TO_CHAR (interval '15h 2m 12s','HH24:MI:SS') -> 15:02:12

TO_DATE(string, format)

Converts *string* to a date using the second argument for the input format. Listed below are the available specifiers for *format* and their meanings.

AM or A.M.

Meridian indicator

AD or A.D.

AD indicator

BC or B.C.

BC indicator

CC

Two-digit century

D

Day of week (1-7)

DD

Day of month (01-31)

DDD

Day of year (001-366)

DAY

Full uppercase day name

Day

Full camel case day name

day

Full lowercase day name

DY

Abbreviated uppercase day name

Dy

Abbreviated camel case day name

dy

Abbreviated lowercase day name

HH or HH12

Hour of day (01-12)

HH24

Hour of day (00-23)

IW

ISO week number of year

J

Julian day number (days since January 1, 4713 BC)

MI

Minute of hour (00-59)

MM

Two-digit month number (01-12)

MON

Abbreviated uppercase month name

Mon

Abbreviated camel case month name

mon

Abbreviated lowercase month name

MONTH

Full uppercase month name

Month

Full camel case month name

month

Full lowercase month name

MS

Milliseconds (000-999)

PM or P.M.

Meridian indicator

Q

Quarter of the year

RM

Roman Numeral month (I-XII)

rm

Lowercase Roman Numeral month (i-xii)

SS

Seconds (00-59)

SSSS

Seconds past midnight (0-86399)

TZ

Uppercase time zone name

tz

Lowercase time zone name

US

Microseconds (000000-999999)

W

Week of month (1-5)

WW

Week of year (1-53)

Y, YY, or YYY

One-, two-, or three-digit years

Y,YYYY

Year with comma

YYYY

Four-digit year

For example:

```
SELECT TO_DATE('05 Dec 2000', 'DD Mon YYYY') -> 2000-12-05
```

TO_NUMBER(string, format)

Converts *string* to a numeric value using the second argument for the input format. For example:

```
SELECT TO_NUMBER('12,454.8-', '99G999D9S') -> -12454.8
```

TO_TIMESTAMP(text, format)

Converts *text* to a timestamp value using the second argument for the input format. (For more on valid format specifiers, see the *TO_DATE* function.) For example:

```
SELECT TO_TIMESTAMP('05 Dec 2000', 'DD Mon YYYY') -> 2000-12-05 00:00:00-08
```

TRANSLATE(text,from,to)

Converts the characters found in *text* that also exist in *from* to the corresponding characters in *to*. For example:

```
SELECT TRANSLATE('foo', 'fo', 'ab') -> 'abb'
```

TRUNC(float8)

Truncates (towards zero). For example:

```
SELECT TRUNC( PI() ) -> 3
```

VARCHAR(string)

Converts *string* to a *VARCHAR*.

WIDTH(box)

Returns the width of *box*. For example:

```
SELECT WIDTH( BOX '((0,0),(3,1))' ) -> 3
```


4.5.5 SQL Server-Supported Functions

This section provides an alphabetical listing of Microsoft SQL Server-supported functions, with examples and corresponding results.

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

APP_NAME()

Returns the application name for the current session, set by the application. For example:

```
SELECT APP_NAME( ) -> 'SQL Enterprise Manager'
```

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATN2(float1, float2)

Returns the angle (in radians) whose tangent is *float1/float2*. For example:

```
SELECT ATN2( 35.175643, 129.44 ) -> 0.265345
```

*BINARY_CHECKSUM(* / expression [,...n])*

Returns the binary checksum for a list of expressions or for a row of a table. This example returns a list of user IDs where the stored password checksum doesn't match the current password's checksum:

```
SELECT userid AS 'Changed' FROM users WHERE NOT password_chksum =  
BINARY_CHECKSUM( password )
```

CHAR(integer_expression)

Converts a numeric ASCII code to a character. For example:

SELECT CHAR(78) -> 'N'

CHARINDEX(substring, string [, start_location])

Returns the position of the first occurrence of a *substring* in a *string*. For example:

SELECT CHARINDEX('he', 'Howdy, there!') -> 9

*CHECKSUM(* / expression [,...n]*)

Returns a checksum (computed over row values or expressions provided). The following example returns a list of user IDs for which the stored password checksum doesn't match the current password's checksum:

**SELECT userid AS 'Changed' FROM users WHERE NOT password_chksum =
BINARY_CHECKSUM(password)**

CHECKSUM_AGG([ALL | DISTINCT] expression)

Returns the checksum of the values in a group. For example:

SELECT CHECKSUM_AGG(BINARY_CHECKSUM(*)) FROM authors -> 67

COALESCE(expression [,...n])

Returns the first non-NULL argument from a list of arguments. For example:

SELECT COALESCE(NULL, 1, 3, 5, 7) -> 1

COL_LENGTH(table, column)

Returns *column* length in bytes. For example:

SELECT COL_LENGTH('authors', 'au_fname') -> 50

COL_NAME(table_id, column_id)

Returns column name, given *table_id* and *column_id*. For example:

SELECT COL_NAME(OBJECT_ID('authors'), 1)

*CONTAINS({column | *}, contains_search_condition)*

Searches *columns* for exact or "fuzzy" matches of the *contains_seach_criteria*. *CONTAINS* is an elaborate function used to perform full-text searches. Refer to the vendor documentation for more information. This example returns all product ID's from the **products** table that contain the words "peanut" and "butter" in close proximity to each other.

**SELECT productid FROM products
WHERE CONTAINS(productname, ' "peanut" NEAR "butter" ')**

CONTAINSTABLE(table, column, contains_search_condition)

Returns a *table* with exact and "fuzzy" matches to *contains_search_condition*. *CONTAINSTABLE* is an elaborate function used to perform full-text searches. The following example returns all product ID's from the **products** table that contain the words "peanut" and "butter" in close proximity to each other:

**SELECT productid FROM products WHERE CONTAINS
(products, productname, ' "peanut" NEAR "butter" ')**

CONVERT(data_type[(length)], expression [, style])

Converts data from one datatype to another. For example:

SELECT CONVERT(VARCHAR(50), CURRENT_TIMESTAMP, 1) -> '06/29/03'

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

COT(number)

Returns the cotangent of *number*. For example:

SELECT COT(3.1415) -> -10792.88993953

DATABASEPROPERTYEX (database, property)

Returns a database option or property. For example:

SELECT DATABASEPROPERTYEX('pubs', 'Version') -> 539

DATALENGTH(expression)

Returns the number of bytes in a character or binary string. For example:

SELECT MAX(DATALENGTH(au_fname)) FROM authors -> 11

DATEADD(datepart, number, date)

Adds a number of *dateparts* (e.g., days) to a datetime value. For example:

SELECT DATEADD(Year, 10, CURRENT_TIMESTAMP) -> 2013-06-29 19:47:15.270

DATEDIFF(datepart, startdate, enddate)

Calculates the difference between two datetime values expressed in the specified *datepart*. For example:

**SELECT DATEDIFF(Day, CURRENT_TIMESTAMP,
DATEADD(Year, 1, CURRENT_TIMESTAMP))**

366

DATENAME(datepart, date)

Returns the name of a *datepart* (e.g., month) of a datetime argument. For example:

SELECT DATENAME(month, GETDATE()) -> 'June'

DATEPART(datepart, date)

Returns the value of a *datepart* (e.g., hour) of a datetime argument. For example:

SELECT DATEPART(year, GETDATE()) -> 2003

DAY(date)

Returns an integer value representing the day of the *date* provided as a parameter. For example:

SELECT DAY('04/15/2004') -> 15

DB_ID([database_name])

Returns a database ID and given name. For example:

SELECT DB_ID() -> 5

DB_NAME(database_id)

Returns the database name. For example:

SELECT DB_NAME(5) -> 'pubs'

DEGREES(numeric_expression)

Converts radians to degrees. For example:

SELECT DEGREES(PI()) -> 180

DIFFERENCE(character_expression, character_expression)

Compares how two arguments sound and returns a number from 0 to 4. A higher result indicates a better phonetic match. For example:

SELECT DIFFERENCE('moe', 'low') -> 3

FILE_ID(file_name)

Returns the file ID for the logical *file_name*. For example:

SELECT FILE_ID('master') -> 1

FILE_NAME(file_id)

Returns the logical filename for the *file_id*. For example:

SELECT FILE_NAME(1) -> 'master'

FILEGROUP_ID(filegroup_name)

Returns filegroup ID for the logical *filegroup_name*. For example:

SELECT FILEGROUP_ID('PRIMARY') -> 1

FILEGROUP_NAME(filegroup_id)

Returns the logical filegroup name for *filegroup_id*. For example:

```
SELECT FILEGROUP_NAME( 1 ) -> 'PRIMARY'
```

FILEGROUPPROPERTY(filegroup_name, property)

Returns the filegroup property value for the specified *property*. For example:

```
SELECT FILEGROUPPROPERTY( 'PRIMARY', 'IsReadOnly' ) -> 0
```

FILEPROPERTY(file, property)

Returns the *file* property value for the specified *property*. For example:

```
SELECT FILEPROPERTY( 'pubs', 'SpaceUsed' ) -> 160
```

FULLTEXTCATALOGPROPERTY(catalog_name, property)

Returns the fulltext catalog properties. For example:

```
SELECT FULLTEXTCATALOGPROPERTY( 'Cat_Desc', 'LogSize' )
```

FULLTEXTSERVICEPROPERTY(property)

Returns the fulltext service level properties. For example:

```
SELECT FULLTEXTSERVICEPROPERTY('IsFulltextInstalled') -> 1
```

FORMATMESSAGE(msg_number, param_value [,... n])

Constructs a message from an existing message in the *SYSMESSAGES* table. (Similar to *RAISEERROR*). For example:

```
sp_addmessage 50001, 1, 'Table %s has %s rows.'
```

```
SELECT FORMATMESSAGE(50001, 'AUTHORS',
```

```
(SELECT COUNT(*) FROM AUTHORS) ) 'Table AUTHORS has 23.'
```

FREETEXT({ column |}, freetext_string)*

Used for a full-text search. Returns rows with *column* that match the meaning, but not exactly the value, of *freetext_string*.

FREETEXTTABLE(table, { column |}, freetext_string [, top_n_by_rank])*

Used for a full-text search. Returns rows from *table* with *column* that match the meaning, but not exactly the value, of *freetext_string*. For example:

```
SELECT * from FREETEXTTABLE (authors, *, 'kev')
```

GETANSINULL([database])

Returns default nullability setting for new columns. For example:

```
SELECT GETANSINULL( ) -> 1
```

GETDATE()

Returns current date and time. For example:

```
SELECT GETDATE( ) -> 2003-06-27 19:26:59.893
```

GETUTCDATE()

Returns the current date as a Coordinated Universal Time (UTC) date. For example:

```
SELECT GETUTCDATE( ) -> 2003-06-28 02:26:46.720
```

GROUPING(column_name)

Returns 1 when a row is added by *CUBE* or *ROLLUP*; otherwise returns 0. For example:

```
SELECT royalty, SUM(advance) 'total advance',  
GROUPING(royalty) 'grp'  
FROM titles  
GROUP BY royalty WITH ROLLUP
```

royalty	total advance	grp
-----	-----	---
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

HOST_ID()

Returns the workstation ID. For example:

```
SELECT HOST_ID( ) -> 216
```

HOST_NAME()

Returns the process host name. For example:

```
SELECT HOST_NAME( ) -> 'PLATO'
```

IDENT_CURRENT(table_name)

Returns the last identity value generated for the specified table. For example:

```
SELECT IDENT_CURRENT('jobs') -> 876
```

IDENT_INCR(table_or_view)

Returns an identity column increment value. For example:

```
SELECT IDENT_INCR('jobs') -> 1
```

IDENT_SEED(table_or_view)

Returns an identity seed value. For example:

```
SELECT IDENT_SEED('jobs') -> 1
```

IDENTITY(data_type [, seed, increment]) AS column_name

Used in a *SELECT INTO* statement to insert an identity column into the destination table. For example:

```
SELECT IDENTITY(int, 1,1) AS ID
```

```
INTO NewTable
```

```
FROM OldTable
```

INDEX_COL(table, index_id, key_id)

Returns an index column name given a table name, index ID, and the sequential number of the column in the index key. For example:

```
SELECT INDEX_COL(OBJECT_ID('authors'), 1, 1) -> NULL
```

INDEXPROPERTY(table_id, index, property)

Returns an index property (such as *FILLFACTOR*). For example:

```
SELECT INDEXPROPERTY(OBJECT_ID('authors'),
```

```
'UPKCL_auind', 'IsPadIndex')
```

```
0
```

ISDATE(expression)

Validates if a character string can be converted to *DATETIME*. For example:

```
SELECT ISDATE(NULL), ISDATE(GETDATE( ))
```

```
0 1
```

IS_MEMBER({group | role})

Returns true or false (1 or 0) depending on whether the user is a member of the specified Windows NT *group* or SQL Server *role*. For example:

```
SELECT IS_MEMBER( 'db_owner' ) -> 0
```

IS_SRVROLEMEMBER(role [,login])

Returns true or false (1 or 0) depending on whether the user is a member of the specified server *role*. For example:

```
SELECT IS_SRVROLEMEMBER( 'sysadmin' ) -> 0
```

ISNULL(check_expression, replacement_value)

Returns the first argument if it is not NULL; otherwise returns the second argument. For example:

SELECT ISNULL(NULL, 'NULL') -> 'NULL'

ISNUMERIC(expression)

Validates if a character string can be converted to *NUMERIC*. For example:

SELECT ISNUMERIC('3.1415'), ISNUMERIC('IRK')

1 0

LEFT(character_expression, integer_expression)

Returns the leftmost *integer_expression* characters of *character_expression*. For example:

SELECT LEFT('Wet Paint', 3) -> 'Wet'

LEN(string_expression)

Returns the number of characters in the expression. For example:

SELECT LEN('Wet Paint') -> 9

LOG(float_expression)

Returns the natural logarithm. For example:

SELECT LOG(PI()) -> 1.1447298858494002

LOG10(float_expression)

Returns the base-10 logarithm. For example:

SELECT LOG10(PI()) -> 0.49714987269413385

LTRIM(character_expression)

Trims leading space characters. For example:

SELECT LTRIM(' beaucoup ') -> 'beaucoup '

MONTH(date)

Returns the month part of the date provided. For example:

SELECT MONTH(GETDATE()) -> 6

NCHAR(integer_expression)

Returns the UNICODE character with the given integer code. For example:

SELECT NCHAR(120) -> 'x'

NEWID()

Creates a new unique identifier of type *UNIQUEIDENTIFIER*. For example:

SELECT NEWID() -> '32B35185-F55E-4FE0-B2C8-B57B35815C12'

NULLIF(expression, expression)

Returns a NULL if the two specified expressions are equivalent. For example:

SELECT NULLIF(5, 5) -> NULL

OBJECT_ID(object)

Returns the object ID of *object*. For example:

SELECT OBJECT_NAME (OBJECT_ID('authors')) -> 'authors'

OBJECT_NAME(object_id)

Returns the object name of an object with the object ID. For example:

SELECT OBJECT_NAME (OBJECT_ID('authors')) -> 'authors'

OBJECTPROPERTY(id, property)

Returns the properties of objects in the current database. For example:

SELECT OBJECTPROPERTY (object_id('authors'),'ISTABLE') -> 1

OPEN {[GLOBAL]cursor_name}| cursor_variable_name}

Opens a local or global cursor.

OPENDATASOURCE(provider_name, init_string)

Makes a connection to a data source without using a linked server name. For examples, refer to the "Loaders" section of the SQL Server User's Guide.

OPENQUERY(linked_server, query)

Queries a remote data source previously configured as a linked server. For an example, refer to the "Loaders" section of the SQL Server User's Guide.

OPENROWSET(provider_name, {datasource; user_id,password | provider_string}, {[catalog.][schema.]object | query})

Queries a remote data source without setting it up as a linked server.

PARSENAME(object_name, object_piece)

Returns the database name, owner name, server name, or object name for the object specified. *Object_piece* is an integer between 1 and 4. For example:

SELECT PARSENAME('pubs..authors', 1) -> 'authors'

SELECT PARSENAME('pubs..authors', 2) -> NULL

SELECT PARSENAME('pubs..authors', 3) -> 'pubs'

SELECT PARSENAME('pubs..authors', 4) -> NULL

PATINDEX('pattern%', expression)

Returns the position of the first occurrence of a pattern in a string. For example:

SELECT PATINDEX('%Du%', 'Donald Duck') -> 8

PERMISSIONS([object_id [, column]])

Returns a numeric value representing a bitmap with the current user's permissions on the specified object or column. For example:

SELECT PERMISSIONS(OBJECT_ID('authors'))&8 -> 8

PI()

Returns the pi constant. For example:

SELECT 2*PI() -> 6.2831853071795862

RADIANS(numeric_expression)

Converts degrees to radians. For example:

SELECT RADIANS(90.0) -> 1.570796326794896600

RAND([seed])

Returns a pseudo-random *FLOAT* type value between and 1. For example:

SELECT RAND(PI()) -> 0.71362925915543995

REPLICATE(character_expression, integer_expression)

Repeats a string a number of times. For example:

SELECT REPLICATE('FOOBAR', 3) -> 'FOOBARFOOBARFOOBAR'

REPLACE(string_expression1, string_expression2, string_expression3)

Performs a search-and-replace on *string_expression1*, replacing each occurrence of *string_expression2* with *string_expression3*. For example:

SELECT REPLACE('Donald Duck', 'Duck', 'Trump') -> 'Donald Trump'

REVERSE(character_expression)

Reverses the characters of a string. For example:

SELECT REVERSE('Donald Duck') -> 'kcuD dlanoD'

RIGHT(character_expression, integer_expression)

Returns the rightmost *integer_expression* characters of *character_expression*. For example:

SELECT RIGHT('Donald Duck', 4) -> 'Duck'

ROUND (number, decimal [,function])

Returns *number* rounded to *decimal* places right of the decimal point. Note that *decimal*, an integer, can be negative to round off digits left of the decimal point. If a nonzero integer is provided for *function*, the return value will be truncated; otherwise the value is rounded. For example:

```
SELECT ROUND(PI( ), 2) -> 3.1400000000000001
```

ROWCOUNT_BIG()

Returns the number of rows affected by the most recent query. (Same as @@*ROWCOUNT*, but returns a *BIGINT* type.) For example:

```
SELECT ROWCOUNT_BIG( ) -> 1
```

RTRIM(character_expression)

Trims trailing space characters from the expression. For example:

```
SELECT RTRIM(' beaucoup ') -> ' beaucoup'
```

SIGN(numeric_expression)

Returns -1 if the argument is negative, 0 if it is zero, and 1 if the argument is positive. For example:

```
SELECT SIGN(-PI( )) -> -1.0
```

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

SOUNDEX(character_expression)

Returns a four-character code based on how the argument string sounds. For example:

```
SELECT SOUNDEX('char') -> 'C600'
```

SPACE(integer_expression)

Returns a string consisting of a given number of space characters. For example:

```
SELECT SPACE(5) -> '     '
```

STATS_DATE(table_id, index_id)

Returns the date and time that index statistics were last updated. For example:

```
SELECT i.name, STATS_DATE(i.id, i.indid)
```

```
FROM sysobjects o, sysindexes i
```

```
WHERE o.name = 'authors' AND o.id = i.id
```

```
UPKCL_auindind 2000-08-06 01:34:00.153
```

```
  aunmind      2000-08-06 01:34:00.170
```

STDEV(expression)

Returns the standard deviation of values in *expression*. For example:

```
SELECT STDEV( qty ) FROM sales -> 16.409201831957116
```

STDEVP(expression)

Returns the standard deviation for the population of values in *expression*. For example:

```
SELECT STDEVP( qty ) FROM sales -> 16.013741264834152
```

STR(number [, length [, decimal]])

Converts *number* to a character string with *length* length and *decimal* decimal places.

STUFF(string1, start, length, string2)

Replaces the *length* characters at position *start* within *string1* with those in *string2*. For example:

```
SELECT STUFF( 'Donald Duck', 8, 4, 'Trump' ) -> 'Donald Trump'
```

SUBSTRING(string, start, length)

Extracts *length* characters from *string* starting at the character in the *start* position. For example:

```
SELECT SUBSTRING( 'Donald Duck', 8, 4 ) -> 'Duck'
```

SUSER_ID([login])

Returns the system user ID of a given login name. Incidentally, this function will always return NULL with SQL Server 2000 or later. Therefore, avoid using this function.

SUSER_SID([login])

Returns the Security ID (SID) for the current user, or for the specified *login*. The SID is returned in binary format. For example:

```
SELECT SUSER_SID('montoyai')
```

```
0x68FC17A71010DE40B005BCF2E443B377
```

SUSER_SNAME([server_user_sid])

Returns the login name for the current user, or for the specified login Security ID (SID). For example:

```
SELECT SUSER_SNAME( ) -> 'montoyai'
```

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TEXTPTR(column)

Returns a pointer to a *TEXT*, *NTEXT*, or *IMAGE* column in *VARBINARY* format. For example:

```
SELECT TEXTPTR(pr_info)
```

```
FROM pub_info WHERE pub_id = '0736'  
  
ORDER BY pub_id  
  
0xFEFF6F0000000005C00000001000100
```

TEXTVALID(table.column, text_ptr)

Returns true or false (1 or 0), depending on whether the provided pointer to a *TEXT*, *NTEXT*, or *IMAGE* column is valid. For example:

```
SELECT pub_id, 'Valid (if 1) Text data'  
      = TEXTVALID ('pub_info.logo', TEXTPTR(logo))  
  
FROM pub_info  
  
ORDER BY pub_id  
  
0736    1  
0877    1  
1389    1  
1622    1  
1756    1  
9901    1  
9952    1  
9999    1
```

TYPEPROPERTY(datatype, property)

Returns information about datatype properties. The *datatype* argument can contain the name of any datatype, and *property* can be a string containing one of the following:

Precision

The precision of the *datatype* is the number of digits or characters that it can store.

Scale

The scale is the number of decimal places for a numeric datatype. A NULL value will be returned if *datatype* is not a numeric datatype. For example:

```
SELECT TYPEPROPERTY('decimal', 'PRECISION') -> 38
```

UNICODE(ncharacter_expression)

Returns the UNICODE code point for the first character of the input parameter. For example:

```
SELECT UNICODE('Hello!') -> 72
```

USER_ID([user])

Returns the user ID for *user* in the current database. If *user* is omitted, then the current user's ID will be returned. For example:

```
SELECT USER_ID( ) -> 2
```

USER_NAME([id])

Returns the current username in the current database. For example:

SELECT USER_NAME() -> 'montoyai'

VAR(expression)

Returns the statistical variance for the values represented by *expression*. For example:

SELECT VAR(qty) FROM sales -> 269.26190476190476

VARP(expression)

Returns statistical variance for the population represented by all values of *expression* in a group. *VARP* is an aggregate function. For example:

SELECT VARP(qty) FROM sales -> 256.43990929705217

YEAR(date)

Returns an integer that is the *YEAR* part of the specified date. For example:

SELECT YEAR(CURRENT_TIMESTAMP) -> 2003



Chapter 5. Database Programming

While SQL plays an important role in standardizing the communication with different RDBMSs, a missing piece still remains for those who want to write database software applications. The missing piece is the database programming Application Programming Interface (API) used to transport SQL statements to and process their results from an RDBMS. While all the database platforms discussed in this book provide their own proprietary interfaces for database application developers, this chapter focuses on two commonly used APIs that provide a consistent interface across database platforms. Specifically, this chapter introduces you to:

ADO.NET

ADO.NET is Microsoft's high-level database programming API on the .NET platform. The ADO.NET API is a collection of .NET interfaces that are accessible from any of the .NET languages. The primary benefits of ADO.NET are ease of use, portability within the .NET platform, XML integration, and access to data sources other than relational databases. The ADO.NET examples covered in this book are written in C#; however, ADO.NET is also available from Visual Basic and the other .NET languages.

JDBC

JDBC, or *Java Database Connectivity*, was developed primarily by Sun Microsystems to be the primary database programming API for the Java language. JDBC is the most popular database programming API for the Java language and offers operating-system portability, reasonable performance for most applications, and a well-documented interface. In addition, drivers for most database systems are typically free. This chapter covers JDBC Version 3.0. For additional information, please browse <http://java.sun.com/jdbc>.

As a quick desk reference, this chapter won't provide all the information needed to develop a large enterprise database application. However, we give you enough to get started by covering components that are common to all database applications, both large and small.

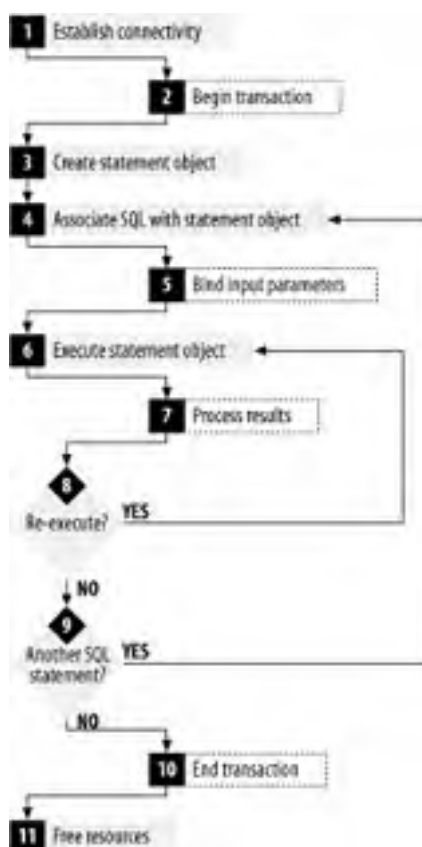
5.1 Database Programming Overview

Developing successful database applications, large and small, involves many steps. Careful thought must be given to application architecture, and especially to the following issues:

- How to map application data, which is typically object-oriented, to a relational database
- How to handle errors gracefully
- How to maximize performance and scalability

A typical database application will require many different SQL statements. The management of so many statements is simplified by the fact that all SQL statements will follow roughly the same pattern of execution within an application. [Figure 5-1](#) is a state diagram showing how SQL statements are prepared, executed, and then processed by a database application when interacting with a relational database system. The state diagram has been broken down into eleven steps, four of which are optional (and are indented in the diagram).

Figure 5-1. Statement execution state diagram



Following are detailed descriptions of each step shown in [Figure 5-1](#):

1. *Establish connectivity*: Establishing connectivity is the first step in every successful database application. It is in this step that the client, or database application, makes a physical connection to the database that will be used to transmit the SQL statements to the database and the results back to the client. The actual physical connection could be over a LAN, WAN, or even a simple logical connection back to the client in cases where the database application and server are running on the same machine. For more information on how to establish connectivity, please see the section [Section 5.2](#) later in this chapter.
2. *Begin transaction (optional)*: A transaction may be begun so that the database changes may be rolled back on failure or committed on success. For more information on transaction control from database programming APIs, please see [Section 5.4](#) later in this chapter.

3. *Create statement object*: Most modern database programming APIs are object-oriented and therefore use an object to represent a SQL statement. There will typically be one statement object per SQL statement executed within the application. The statement object holds the state information required to execute the SQL statement, such as the SQL statement itself and input parameters when they exist.
4. *Associate SQL with statement object*: After the statement object has been created, it needs to have a SQL statement assigned to it. Once this is done, the statement object may be executed.
5. *Bind input parameters (optional)*: While not part of the ANSI SQL standard, the ability to bind parameters to "placeholders" within a SQL statement is supported by all the database platforms covered in this book. If the SQL statement has placeholders for input parameters, the statement object will need to have a program variable associated with each input parameter. If the SQL statement contains no input parameters, then this step may be skipped. Input parameters are useful for optimizing performance when the same SQL statement is re-executed many times, because the server-side parsing of the statement only needs to be done once prior to the first execution. Another reason to use input parameters is to embed binary data, such as *BLOB* data, into SQL statements such as *INSERT* and *UPDATE* statements.
6. *Execute statement object*: After the statement object has been successfully created and initialized with a SQL statement, the statement object can be executed. This step executes the SQL statement on the database server.
7. *Process Results (optional)*: After the database server returns a result set, the application may process the results. This step is optional, since it is typically not required for statements that insert or update data in the database.
8. *Re-execution*: If the same statement needs to be re-executed to retry on a failure or to execute again with different values for the input parameters, the application can return to Step 6. If the application has no need to re-execute the same statement, it moves on to Step 9.
9. *Execute another SQL statement*: If the application needs to execute a different SQL statement and can reuse the statement object, then the application can return to Step 4; if not, it can move to Step 10.
10. *End transaction (optional)*: Assuming a transaction was begun in step 2, the transaction is now either committed or rolled back. If the transaction is rolled back, then all changes to the database made by the statement object will be removed from the database.
11. *Free resources*: After successfully executing the statement and processing any results, the client- and server-side resources need to be freed for use by other applications.

The remaining sections in this chapter provide examples of how to use the ADO.NET and JDBC database APIs to build applications that follow the steps outlined in [Figure 5-1](#).

5.2 Opening a Database Connection

Before interacting with a database, an application must first establish a connection to the database server. The APIs covered in this book abstract the low-level connectivity details into a few simple object-oriented classes, making it easier to focus on the database application instead of protocols and network topology.

5.2.1 Opening an ADO.NET Database Connection

Opening a connection with ADO.NET requires instantiating a connection object with a properly formatted connection string and then invoking the `Open` method on the connection object. The connection object can be an `OdbcConnection`, `SqlConnection`, or an `OleDbConnection`. The `OdbcConnection` is designed for any ODBC datasource, and the `OleDbConnection` type will work with any OLE DB Provider. For the highest performance data access, use connection objects specifically tuned to the specific database platform, such as `SqlConnection` for Microsoft SQL Server. Following is the syntax for creating a `Connection` object in ADO.NET:

```
{Odbc|OleDb|Sql}Connection connection =  
    new {Odbc|OleDb|Sql}Connection(connection_string);  
connection.Open( );
```

The format of the connection string is the same for all of the connection types. The format is a string of key/value pairs delimited by semicolons. For example:

```
key1=value1; key2=value2; key3=value3; ...
```

While the format is the same for every connection type, the keys and values are quite different. Tables [Table 5-1](#) through [Table 5-3](#) list the attributes for the three connection types listed above. Many database platforms support additional attributes that can also be set through the connection string. For a list of these attributes, please consult the appropriate database vendor documentation.

Following are examples of two connection strings for an `OdbcConnection`:

```
DSN=MyOracleDSN; UID=scott; PWD=tiger;  
DRIVER={SQL Server};SERVER=(local);UID=sa;PWD=;DATABASE=pubs;
```

The first string connects to a Data Source Name, or DSN, with the name **MyOracleDSN** using the username **scott** and the password **tiger**.

The second string connects using the **SQL Server** driver to a database named **pubs** on the local server. The username is **sa**. The password is blank, indicating to the driver that the password is not required for the **sa** user.

Following is an example of a connection string for an `OleDbConnection`, which connects to an Oracle9i data source using the MSDAORA OLE DB provider, the use name **scott**, and the password **tiger**.

```
Provider=MSDAORA;Data Source=Oracle9;User ID=scott;Password=tiger;
```

Finally, here is an example of a connection string for a `SqlConnection`, which connects to a SQL Server data source on the local server:

```
Server=(local);UID=sa;PWD=;DATABASE=pubs;Connection Timeout=60;
```

In Tables [Table 5-1](#) through [Table 5-3](#), you'll find synonyms for some keywords. For example, you can use "DSN" and "Data Source Name" interchangeably in your code.

Table 5-1. Connection string attributes for OdbcConnection

Keyword	Notes
---------	-------

Data Source Name DSN	The DSN, FILEDSN, or DRIVER attribute must be provided to connect. The DSN is the Data Source Name known by the client's driver manager. The advantage of using a DSN is that the DSN can be changed to point to a different database platform while the database applications are still running.
User ID UID	This value is set to the user identifier authorized to open a new connection.
Password PWD	The password for the user. If the user doesn't have a password for the data source, then this property should still be supplied with an empty value.
DSN_File Name FILEDSN	Similar to the DSN, the FILEDSN is a file, typically with a DSN extension that contains the attributes for the connection object to establish connectivity. Even when using a FILEDSN, the connection may still need the password provided, since passwords aren't stored within DSN files.
Driver Name DRIVER	A Driver may be explicitly used instead of a DSN or FILEDSN. The drawback of using a driver directly is that the application won't be isolated from changes made to the driver and will require slight modifications to target a different database platform.
SAVEFILE	When the SAVEFILE attribute is set to a legal filename, the connection attributes will be saved to the file once successful connectivity is established. This option is only available when using DRIVER and FILEDSN connectivity methods.

Table 5-2. Connection string attribute for OleDbConnection

Keyword	Notes
Provider	The name of the provider: this is the only required attribute. The specified provider may require additional attributes.

Table 5-3. Connection string attributes for SqlConnection

Keyword	Notes
Application Name	The name of the application that will be displayed from server management utilities.
AttachDBFilename Extended properties Initial File Name	The pathname to an attachable database.
Connect Timeout Connection Timeout	Number of seconds to wait before the connection attempt is aborted. The default is 15 seconds.
Connection Lifetime	Time in seconds the connection can remain in the connection pool, when pooling is enabled.
Connection Reset	Specifies whether or not the connection is reset when it is reused from the connection pool. Default is true.
Current Language	The language the connection session should use.
Data Source Server Address Addr Network Address	The SQL Server instance name or network address.
Initial Catalog Database	The name of the database.
Integrated Security Trusted_Connection	Set to true or <code>sspi</code> for secure connections. The default is false.

Max Pool Size	The maximum number of connections allowed in the connection pool at a time. The default is 100.
Min Pool Size	The minimum number of connections to keep in the connection pool at a time. The default is 0.
Network Library Net	The network library to use in establishing the connection. Valid settings are <code>dbnmpntw</code> for Named Pipes, <code>dbmsrpcn</code> for Multiprotocol, <code>dbmsadsn</code> for AppleTalk, <code>dbmsgnet</code> for VIA, <code>dbmsipcn</code> for Shared Memory, <code>dbmsspxn</code> for IPX/SPX, and <code>dbmssocn</code> for TCP/IP. The default is <code>dbmssocn</code> for TCP/IP.
Packet Size	The network packet size in bytes; the default is 8,192.
Password Pwd	The user's password.
Persist Security Info	Determines if security-sensitive connection properties, such as the password, are stored within the connection object after a connection has been attempted or completed. The default is false.
Pooling	Determines if connection pooling should be used for the connection. The default is true.
User ID	The user's login name.
Workstation ID	The name of the computer connecting to the database.

5.2.2 Opening a JDBC Database Connection

Following is the syntax for registering a driver with the JDBC Driver Manager and then opening a database connection:

```
Class.forName(driver_name);  
  
Connection connection = DriverManager.getConnection(connect_string,  
                                                username, password);
```

The first step in establishing JDBC connectivity is to instruct the Java Virtual Machine (JVM) class loader to load the appropriate JDBC driver.

The most common method of loading the driver into the class loader is to use the static `forName` method of the `Class` class. This method can provide applications with greater flexibility in changing database platforms by having the Java Virtual Machine (JVM) load the database driver at runtime:

```
Class.forName(" driver_name ");
```

After the database driver has been loaded, the application can establish connectivity by invoking the static `getConnection` method on the JDBC `DriverManager` class. The `getConnection` method takes three arguments: a connection string, username, and password:

```
Connection connection = DriverManager.getConnection(connect_string,  
                                                username,      password);
```

The connection string follows the following JDBC URL naming scheme:

```
jdbc:subprotocol.subname
```

Following are examples of how to connect to the different vendors covered in this book using JDBC.

5.2.2.1 DB2

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");  
  
Connection connection = DriverManager.getConnection(  
    "jdbc:db2:DATABASE", "user", "passwd" );
```

5.2.2.2 MySQL

```
Class.forName("org.gjt.mm.mysql.Driver");  
Connection connection = DriverManager.getConnection(  
    "jdbc:mysql://127.0.0.1:3306/DATABASE",  
    "user", "passwd" );
```

5.2.2.3 PostgreSQL

```
Class.forName("org.postgresql.Driver");  
Connection connection = DriverManager.getConnection(  
    "jdbc:postgresql://127.0.0.1:5432/DATABASE",  
    "user", "passwd" );
```

5.2.2.4 Oracle

```
Class.forName("oracle.jdbc.driver.OracleDriver");  
Connection connection = DriverManager.getConnection(  
    "jdbc:oracle:thin:@myserver",  
    "scott", "tiger" );
```

5.2.2.5 SQL Server

```
Class.forName(  
    "com.microsoft.jdbc.sqlserver.SQLServerDriver"  
);  
Connection connection = DriverManager.getConnection(  
    "jdbc:microsoft:sqlserver://SERVER:1433;" +  
    "DatabaseName=pubs;",  
    "user", "passwd" );
```

5.3 Closing a Database Connection

Once an application is finished using an open connection, that connection should be closed. Some APIs automatically close connections when the application exits; however, it is best to explicitly close the connection as soon as the application is finished accessing the database to allow other applications access to precious connection resources.



Calling the appropriate close methods when finished with an ADO.NET or JDBC object is essential to freeing precious database resources used by the application. Both ADO.NET and Java offer very few guarantees when objects are finalized; therefore, it is common for those applications that don't close objects explicitly to run out of database resources (such as connections, statement handles, and memory).

5.3.1 Closing an ADO.NET Database Connection

To close an *ADO.NET* connection, invoke the `Close` method on the connection object:

```
connection.Close( );
```

5.3.2 Closing a JDBC Database Connection

To close a *JDBC* connection, invoke the `close` method on the connection object:

```
connection.close( );
```

5.4 Managing Transactions

Most database programming APIs provide methods of controlling transactions, setting savepoints, and modifying isolation levels. This section covers the mechanisms for controlling transactions from the ADO.NET and JDBC APIs covered in this chapter.

5.4.1 Beginning a Transaction

Beginning a transaction is the first step in guaranteeing atomicity when executing multiple SQL statements. After beginning a transaction, the transaction can be committed to make the changes performed by the executed SQL statements permanent or the transaction can be rolled back to abort the changes and leave the database unchanged.

5.4.1.1 Beginning an ADO.NET transaction

To begin a transaction using ADO.NET, invoke the `BeginTransaction` method on a connection object. A `Transaction` object is returned that can be used in the creation of ADO.NET `Command` objects that execute within the same transaction. To execute a `Command` object within the new transaction, you must first attach the `Transaction` object to the `Command` object's `Transaction` property. Following is the syntax for starting a transaction by creating a `Transaction` object in ADO.NET:

```
{Odbc|OleDb|Sql}Transaction transaction =  
    connection.BeginTransaction([IsolationLevel.  
        {Chaos | ReadCommitted | ReadUncommitted | RepeatableRead |  
        Serializable | Unspecified}]);  
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
statement.Transaction = transaction;
```

When creating the `Transaction` object, you can optionally specify the *isolation level* that is used for the transaction. The isolation level controls how much (or little) your database transactions are insulated from the effects of other transactions. The available isolation levels are:

Chaos

Pending changes of other transactions with higher isolation levels (`ReadCommitted`, `ReadUncommitted`, `RepeatableRead`, or `Serializable`) cannot be changed.

ReadUncommitted

The lowest ANSI standard isolation level, unprotected against dirty reads, non-repeatable reads, and phantom records.

ReadCommitted

The next highest ANSI standard isolation level above `ReadUncommitted`, protects against dirty reads.

RepeatableRead

The next highest ANSI standard isolation level above `ReadCommitted`, protects against dirty reads and nonrepeatable reads.

Serializable

The highest ANSI standard isolation level. Protects against dirty reads, non-repeatable reads, and phantom records.

Unspecified

Isolation level cannot be determined. This option is not typically passed into the `BeginTransaction` method; its purpose is to provide an Unknown state for the `Connection` object's `IsolationLevel` property. If the `Transaction` has this `IsolationLevel`, even after being set to another value, then it is safe to assume that the database does not support the `IsolationLevel` that you requested.

For more information about isolation levels, see [Programming Tips and Gotchas](#) under [SET TRANSACTION Statement](#).



The isolation level used during SQL statement execution can have a significant impact on the performance and scalability of a database application, as well as on the RDBMS itself. When deciding which isolation level to use, choose the lowest level that provides the appropriate isolation guarantees for the database application.

5.4.1.2 Beginning a JDBC transaction

JDBC starts a connection in `AUTO COMMIT` mode, which is a setting that automatically commits changes made by each SQL statement when it executes. To gain control over when changes are committed, the `AUTO COMMIT` setting must be turned off. JDBC's mechanism of beginning a transaction is subtle compared to ADO.NET, which uses a `BeginTransaction` method to begin a transaction. In JDBC, a transaction is started as soon as the `AUTO COMMIT` mode is turned off or just after the transaction has been committed or rolled back. Committing or rolling back the transaction doesn't turn the `AUTO COMMIT` mode back on, so multiple commits or rollbacks will need to be executed to commit/roll back multiple units of work until the connection is closed or `AUTO COMMIT` is explicitly turned back on. To turn off the `AUTO COMMIT` mode, execute the `setAutoCommit` method of the JDBC `Connection` object with a false argument:

```
connection.setAutoCommit( false );
```



Beware that invoking `setAutoCommit` to turn `AUTO COMMIT` mode on or off when a transaction has already begun will automatically commit that transaction.

JDBC also supports different isolation levels through the JDBC `Connection` interface. To set the isolation level used for new transactions, invoke the `setTransactionIsolation` method with one of the following isolation levels:

TRANSACTION_NONE

Indicates transactions are not supported on the connection. This value is not typically used to set the isolation level, but to query the connection for transaction support with the `getTransactionIsolation` method on the `Connection` interface.

TRANSACTION_READ_UNCOMMITTED

The lowest ANSI standard isolation level, unprotected against dirty reads, nonrepeatable reads, and phantom records.

TRANSACTION_READ_COMMITTED

The next highest ANSI standard isolation level above `TRANSACTION_READ_UNCOMMITTED`. Protects against dirty reads.

TRANSACTION_REPEATABLE_READ

The next highest ANSI standard isolation level above `TRANSACTION_READ_COMMITTED`. Protects against dirty reads and nonrepeatable reads.

TRANSACTION_SERIALIZABLE

The highest ANSI standard isolation level. Protects against dirty reads, nonrepeatable reads, and phantom records.

For a more detailed explanation of transaction isolation levels, see [Programming Tips and Gotchas](#) under [SET TRANSACTION Statement](#).

Following is example code setting the isolation level in JDBC:

```
connection.setIsolationLevel( Connection.TRANSACTION_SERIALIZABLE );
```



The isolation level used on the connection can have a significant impact on the performance and scalability of a database application, as well as on the RDBMS itself. When deciding which isolation level to use, choose the lowest level that provides the appropriate isolation guarantees for the connection.

5.4.2 Committing a Transaction

Committing a transaction is a way to explicitly close the transaction and make its database modifications permanent.

5.4.2.1 Committing an ADO.NET transaction

To commit a transaction using ADO.NET, invoke the `Commit` method on the `Transaction` object:

```
transaction.Commit( );
```

ADO.NET is more object-oriented than many database-programming APIs, as can be seen from the encapsulation of a transaction into a unique object type. This has the drawback of making the methods for explicitly ending a transaction more difficult to find than those to begin a transaction, since they are on different object types. Remember that the method to begin a transaction is on the `Connection` object, and the methods to commit or roll back a transaction are found on the `Transaction` object.

5.4.2.2 Committing a JDBC transaction

Invoking the `commit` method on a JDBC `Connection` object commits a transaction within JDBC and begins a new transaction:

```
connection.commit( );
```

If the connection is in `AUTO COMMIT` mode, you will then get a Java exception of type `SQLException` because the connection will not have a pending transaction.

5.4.3 Rolling Back a Transaction

Rolling back a transaction is a way to close the transaction explicitly and to discard any database modifications since the transaction was started.

5.4.3.1 Rolling back an ADO.NET transaction

To roll back a transaction using ADO.NET, invoke the `Rollback` method on the `Transaction` object:

```
transaction.Rollback( );
```

After rolling back a `Transaction` object, the `Dispose` method should be invoked to free any resources held by the

transaction:

```
transaction.Dispose( );
```

5.4.3.2 Rolling back a JDBC transaction

Invoking the `rollback` method on a JDBC `Connection` object rolls back a transaction within JDBC and begins a new transaction:

```
connection.rollback( );
```

If the connection is in `AUTO COMMIT` mode, then a Java exception will be thrown of `SQLException` type.



5.5 Executing Statements

The primary purpose of programming to a specific database API is to execute SQL statements. The next section covers the steps required to successfully execute a simple SQL statement, such as *INSERT* or *UPDATE*, that does not return results. The section following focuses on the more complicated task of executing statements that do return results.

5.5.1 Executing an ADO.NET Statement

The following code fragment provides the syntax for executing a SQL *INSERT* statement using ADO.NET:

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
statement.CommandText = "INSERT INTO authors(au_id, au_lname, " +  
    "          au_fname, contract) " +  
    "VALUES ('xyz', 'Brown', 'Emmit', 1)";  
int rowsInserted = statement.ExecuteNonQuery( );  
statement.Close( );
```

To execute a statement using ADO.NET, take the following steps:

1. Create an ADO.NET **Command** object. **SqlCommand** is used for access to Microsoft SQL Server, **OdbcCommand** is used for ODBC data sources, and **OleDbCommand** for OLE-DB data sources.

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );
```

2. After creating the appropriate **Command** object, a SQL statement needs to be associated with the **Command** before it can be executed.

```
statement.CommandText = "INSERT INTO authors(au_id, au_lname, " +  
    "          au_fname, contract) " +  
    "VALUES ('xyz', 'Brown', 'Emmit', 1)";
```

3. After assignment of the SQL statement, the **Command** object is ready for execution.

```
int rowsInserted = statement.ExecuteNonQuery( );
```

There are three methods that can be used to execute ADO.NET **Command** objects:

- a. **ExecuteReader**: Executes SQL statements that return rows, such as *SELECT* statements. The return value is an *ADO.NET DataReader* object.
- b. **ExecuteNonQuery**: Executes SQL statements that do not typically return result sets, such as *INSERT*, *DELETE*, or *UPDATE* statements. The return value is an integer value equal to the number of rows affected by the statement execution.
- c. **ExecuteScalar**: Executes SQL statements that return a single value, such as statements containing a single aggregate function.
- d. After the **Command** object is executed, it should be explicitly closed to free up resources that have been allocated to execute statements and to communicate with the database server. To free up these resources, invoke the **Command** object's **Dispose** method:

```
statement.Dispose( );
```



The resources held by ADO.NET **Statement** objects are freed using the **Dispose** method, which differs from the **Close** method used by all other ADO.NET types.



5.5.2 Executing a JDBC Statement

The following code fragment provides the syntax for executing a SQL *INSERT* statement using JDBC:

```
java.sql.Statement statement = connection.createStatement( );  
  
int rowsInserted = statement.executeUpdate(  
    "INSERT INTO authors(au_id, au_lname, au_fname, contract) " +  
    " VALUES ('xyz', 'Brown', 'Emmit', 1)");  
  
statement.close( );
```

1. Executing a SQL statement first requires the creation of a JDBC **Statement** object. A **Statement** object is produced by invoking the `createStatement` method on a valid JDBC **Connection** object:

```
java.sql.Statement statement = connection.createStatement( );
```

2. After a **Statement** object has been created, a SQL statement can be executed by invoking one of the execute methods found on the **Statement** object. For non-query statements, the `executeUpdate` method is the best method to use:

```
statement.executeUpdate("INSERT INTO authors(au_id, au_lname, " +  
    "au_fname, contract) VALUES " +  
    "('xyz', 'Brown', 'Emmit', 1)");
```

The `executeUpdate` method returns an `int` value indicating the number of rows inserted, updated, or deleted during the execution of the SQL statement. If the execution fails, you will get a **SQLException**.

3. After the **Statement** object has been executed, it may be executed again, or freed by invoking the `close` method.

```
statement.close( );
```



5.6 Retrieving Data

A typical database program retrieves data from a database server and processes it. These programs will execute *SELECT* statements following a process similar to that described in the previous section. The difference between executing a *SELECT* statement as opposed to a statement that does not return results is that with *SELECT* you must execute additional code to process the results returned by *SELECT*.

5.6.1 Retrieving Data Using ADO.NET

The following C# code fragment executes a SQL *SELECT* statement that returns author names from the **authors** table and then iterates through the results one row at a time, printing out each author's name. The code is explained in the detailed steps that follow:

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
  
statement.CommandText = "SELECT au_fname, au_lname FROM authors";  
  
{Odbc|OleDb|Sql}DataReader resultSet = statement.ExecuteReader( );  
  
while( resultSet.Read( ) )  
{  
    String fname = "NULL";  
    String lname = "NULL";  
  
    if( !resultSet.IsDBNull( 0 ) ) fname = resultSet.GetString( 0 );  
    if( !resultSet.IsDBNull( 1 ) ) lname = resultSet.GetString( 1 );  
  
    System.Console.WriteLine( lname + ", " + fname );  
}  
  
resultSet.Close( );  
statement.Close( );
```



Column ordinals are all zero-based (the first column is 0, the second is 1, etc.) in ADO.NET, which is different from the one-based ordinals used by JDBC.

To execute a SQL query and process the results using ADO.NET, take the following steps:

1. Create the **Command** object that will be used to execute the *SELECT* statement and attach a *SELECT* statement to it:

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
  
statement.CommandText = "SELECT au_fname, au_lname FROM AUTHORS";
```

2. Invoke the **ExecuteReader** method on the **Command** object, creating a new **DataReader** object.

```
{Odbc|OleDb|Sql}DataReader resultSet = statement.ExecuteReader( );
```

3. Iterate through each row in the result set. After execution of the *SELECT* statement, retrieve rows using the **DataReader's Read** method. If you expect a multiple row result, you should invoke the **Read** method from within a while loop:

```
while( resultSet.Read( ) )
{
```

4. Having fetched a row of data into the **DataReader** object, the column data can be extracted using the **DataReader's Get** methods. Before calling the **Get** methods, we check to see if the data is NULL using the **DataReader's IsDBNull** method. If the value returned by the **IsDBNull** method is true, then the value of the string will be NULL. There are many **Get** methods for each type of column. Using the appropriate datatype for the column is important, because a conversion is not always possible. For a list of available **Get** methods and the ANSI SQL types they should be used with, please see [Table 5-4](#).

```
String fname = "NULL";

String lname = "NULL";

if( !resultSet.IsDBNull( 0 ) ) fname = resultSet.GetString( 0 );

if( !resultSet.IsDBNull( 1 ) ) lname = resultSet.GetString( 1 );
```

When creating an ADO.NET program that fetches data that can contain NULL values, it is always safest to check the return value of the **DataReader** object's **IsDBNull** method prior to extracting a value.

5. The **Read** method of **DataReader** returns false when all the data has been read from the database server. At that point, the **Close** methods on the **Command** object and **DataReader** should be invoked to free up the resources used internally to process the statement.

```
resultSet.Close( );
```

```
statement.Close( );
```

Table 5-4. ADO.NET DataReader Get methods

Method name	Description
GetBoolean(int i)	Returns the value from the <i>i</i> th column as a Boolean value, where <i>i</i> is the zero-based column number.
GetByte(int i)	Returns the value from the <i>i</i> th column as a single-byte value, where <i>i</i> is the zero-based column number. No conversion will be done if the data in the column exceeds one byte and an InvalidCastException object will be thrown.
GetBytes(int i, long dataIndex, byte[] buffer, int bufferIndex, int length)	Returns the value from the <i>i</i> th column as a binary value, where <i>i</i> is the zero-based column number, <i>dataIndex</i> is the offset in the column value to start reading, <i>buffer</i> is the byte array to copy the data into, <i>bufferIndex</i> is the offset into <i>buffer</i> at which to start copying data, and <i>length</i> is the maximum length to copy into <i>buffer</i> .
GetChar(int i)	Returns a single char value from the <i>i</i> th column. For use with character type columns, where <i>i</i> is the zero-based column number.
GetChars(int i, long dataIndex, char[] buffer, int bufferIndex, int length)	Returns a string of characters from the <i>i</i> th column. For use with character type columns, where <i>i</i> is the zero-based column number, <i>dataIndex</i> is the offset in the column value to start reading, <i>buffer</i> is the char array to copy the data into, <i>bufferIndex</i> is the offset into <i>buffer</i> at which to start copying data, and <i>length</i> is the maximum length to copy into <i>buffer</i> .
GetDataTypeName(int i)	Gets a String value that contains the name of the column's datatype, where <i>i</i> is the zero-based column number.
GetDateTime(int i)	Returns a DateTime value from the <i>i</i> th column. For use with temporal type columns, where <i>i</i> is the zero-based column number.
GetDecimal(int i)	Returns a single Decimal value from the <i>i</i> th column. For use with numeric type columns, where <i>i</i> is the zero-based column number.
GetDouble(int i)	Returns a single double value from the <i>i</i> th column. For use with double precision type columns, where <i>i</i> is the zero-based column number.
GetFloat(int i)	Returns a single float value from the <i>i</i> th column. For use with floating-point columns, where <i>i</i> is the zero-based column number.
GetInt{16,32,64}(int i)	Retrieves data from integer columns. The precision of the return value is encoded within the function names. Use GetInt16 for a 16-bit signed short integer, GetInt32 for a 32-bit signed int , and GetInt64 for a 64-bit signed long .

<code>GetName(int i)</code>	Returns a String value containing the name of the <i>i</i> th column, where <i>i</i> is the zero-based column number.
<code>GetOrdinal(string name)</code>	Returns an int value containing the ordinal value of the column with name matching the <i>name</i> argument.
<code>GetString(int i)</code>	Returns a single String value from the <i>i</i> th column. For use with character type columns, where <i>i</i> is the zero-based column number.

Beyond the **Get** methods, there are three other methods on the **DataReader** type that are frequently used when processing data from a query. Those three methods are listed in [Table 5-5](#).

Table 5-5. Frequently used ADO.NET DataReader methods

Method name	Description
<code>Close()</code>	Closes the DataReader object, freeing up resources held by the instance.
<code>IsDBNull(int i)</code>	Returns true if the specified column is NULL, otherwise it returns false, where <i>i</i> is the zero-based column number.
<code>Read()</code>	Fetch the next row if one is available, and return true, otherwise return false.

5.6.2 Retrieving Data Using JDBC

The following Java code fragment executes a SQL *SELECT* statement that returns author names from the **authors** table and then iterates through the results one row at a time, printing out each author's name:

```
java.sql.Statement statement = connection.createStatement();

java.sql.ResultSet result =

    statement.executeQuery("SELECT au_fname, au_lname FROM authors");

while( result.next() ) {

    String fname = result.getString( 1 );

    if( result.wasNull() ) fname = "NULL";

    String lname = result.getString( 2 );

    if( result.wasNull() ) lname = "NULL";

    System.out.println( lname + ", " + fname );

}

result.close();

statement.close();
```



Column ordinals are all one-based (first column is 1, second is 2, etc.) in JDBC, which is different from the zero-based ordinals used by ADO.NET.

5.6.2.1 Use the following steps to execute query statements in JDBC:

1. Create a JDBC **Statement** object by invoking the `createStatement` method on a valid **Connection** object:

```
java.sql.Statement statement = connection.createStatement();
```

- The query is executed by invoking one of the `execute` methods on the `Statement` object. Result sets from query statements are processed by JDBC `ResultSet` objects, which are returned from the `executeQuery` method of JDBC `Statement` objects.

```
java.sql.ResultSet result =
    statement.executeQuery("SELECT au_fname, au_lname FROM authors" );
```

- After creating a `ResultSet` object, you can iterate through one row at a time by invoking the `next` method. The `next` method returns a Boolean value, true for each row in a result set and false after all rows have been iterated through. It is common to invoke the `next` method within a `while` loop to process the rows one-by-one. The `ResultSet` does not begin on the first row of the result, so you must invoke the `next` method to advance to the first row before calling any of the `get` methods.

```
while( result.next( ) ) {
```

- To retrieve column data from the rows within a result set, invoke the appropriate `get` method on the `ResultSet` object. For a list of the most common `get` methods, check [Table 5-6](#). Note that the column data is checked for a NULL value after the `get` method has been invoked, since the nullness of a value can't be determined from the value returned by the `get` methods.

```
String fname = result.getString("au_fname");
if( result.isNull( ) ) fname = "NULL";
String lname = result.getString("au_lname");
if( result.isNull( ) ) lname = "NULL";
```

When creating a JDBC program that fetches data from nullable columns, it is always safest to check the value returned from a `ResultSet` object for a NULL value using the `isNull` method.



The JDBC `get` methods that return object types will return Java NULL values when returning database NULL values. However, this does not apply to non-object types such as `getInt()`, which returns zero in the case of a database NULL. For this reason, this chapter uses the verbose `isNull()` method to test the value for a NULL.

- Free the resources held by the result set and statement objects. When finished with the `ResultSet` and `Statement` objects, invoke their `close` methods so that database resources can be freed.

```
result.close( );
statement.close( );
```

Table 5-6. JDBC ResultSet get methods

Method name	Description
<code>getBlob({int i/String name})</code>	Retrieves a <code>Blob</code> value from <code>BLOB</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getBoolean({int i/String name})</code>	Retrieves a <code>boolean</code> value from <code>BOOLEAN</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getByte({int i/String name})</code>	Retrieves a <code>byte</code> value from <code>CHARACTER</code> or <code>BINARY</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getBytes({int i/String name})</code>	Retrieves a <code>byte[]</code> value from <code>BINARY</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getClob({int i/String name})</code>	Retrieves a <code>Clob</code> value from <code>CLOB</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getDate({int i/String name})</code>	Retrieves a <code>Date</code> value from <code>TEMPORAL</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getDouble({int i/String</code>	Retrieves a <code>double</code> value from <code>DOUBLE PRECISION</code> type columns, where <code>i</code> is the one-based

<i>name</i>)	column ordinal and <i>name</i> is the name of the column.
getFloat({int <i>i</i> String <i>name</i> })	Retrieves a float value from <i>REAL</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
getInt({int <i>i</i> String <i>name</i> })	Retrieves an int value from <i>INTEGER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
getLong({int <i>i</i> String <i>name</i> })	Retrieves a long value from <i>INTEGER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
getRow()	Returns the current row number.
getShort({int <i>i</i> String <i>name</i> })	Retrieves a short value from <i>INTEGER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
getString({int <i>i</i> String <i>name</i> })	Retrieves a String value from <i>CHARACTER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
getTime({int <i>i</i> String <i>name</i> })	Retrieves a Time value from <i>TEMPORAL</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
getTimestamp({int <i>i</i> String <i>name</i> })	Retrieves a Timestamp value from <i>TEMPORAL</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.

Beyond the **get** methods, there are three other methods on the **ResultSet** type that are frequently used when processing data from a query. Those three methods are listed in [Table 5-7](#).

Table 5-7. Frequently used JDBC ResultSet methods

Method name	Description
close()	Closes the ResultSet object, freeing up resources held by the instance.
next()	Advances the ResultSet object to the next available row and returns true. If no rows remain false will be returned.
wasNull()	Returns true if the last column returned with a get method contained a database NULL value, returns false otherwise.

5.7 Bound Parameters

One of the easiest ways to optimize a slow database application that executes many similar SQL statements is to parameterize the most commonly used SQL statements. Parameterization is a way of reusing a SQL statement by writing it with placeholders for frequently changing values. This provides two major benefits: the potential for reduced server roundtrips and better processing efficiency on the database server, since the server doesn't need to parse, plan, and optimize the execution of frequently used SQL statements on every execution.

Parametrized statements are useful in using string or binary values within SQL statements that contain unfriendly characters, such as quote marks or terminal NULL characters that indicate the end of a string. Usage in this way provides security benefits to a database application where user input is used within statements. Not binding a parameter coming from an end user could allow clever modification of the statement in a way that may divulge secret information from the database.



Since there is a slight cost involved in setting up a parameterized statement, it's good practice to only parameterize statements that will be executed at least three times prior to the statement closing. If a statement is only executed once prior to freeing the statement, the costs of preparing the statement and binding the parameters could result in a performance net loss to the application.

5.7.1 ADO.NET Bound Parameters

The following C# code fragment executes a SQL *INSERT* statement that adds new sales to the **sales** table in the **pubs** database. The *INSERT* statement is parameterized to provide better performance, since the statement object needs to be parsed only once on the server.

```
// Create a Command object for the SQL statement

Statement statement = connection.CreateCommand( );

statement.CommandText =

    "INSERT INTO SALES(stor_id,

        ord_num,

        ord_date,

        qty,

        payterms,

        title_id) " +

    "VALUES(@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id)";

//Prepare the statement on the server

statement.Prepare( );

// Declare parameters that will be bound

{Odbc|OleDb|Sql}Parameter stor_id, ord_num, ord_date,

    qty, payterms, title_id;

stor_id = statement.Parameters.Add( "@stor_id", DbType.String );

ord_num = statement.Parameters.Add( "@ord_num", DbType.String );

ord_date = statement.Parameters.Add( "@ord_date", DbType.DateTime);
```

```
qty = statement.Parameters.Add( "@qty", DbType.Int16 );
payterms = statement.Parameters.Add( "@payterms", DbType.String );
title_id = statement.Parameters.Add( "@title_id", DbType.String );

while( GetNextSale(stor_id, ord_num, ord_date, qty, payterms, title_id )
{
    // Execute the statement
    int result = statement.ExecuteNonQuery( );
    if( result != 1 )
    {
        // If result isn't 1, then the insert failed.
        System.Console.WriteLine( "The INSERT failed." );
        break;
    }
}
```

5.7.1.1 Use the following steps to execute statements with bound parameters in ADONET:

1. As done in previous sections, we create an ADO.NET **Command** object and assign a SQL statement to it. The difference for using bound parameters within the statement is in the **VALUES** clause of the **INSERT** statement. Contained within the **VALUES** clause are six named placeholders for the parameters that will be later bound to the **Command** object. In ADO.NET the placeholders begin with the @ symbol and are followed by an identifier that is unique amongst all placeholders in the statement.

```
Statement statement = connection.CreateCommand( );

statement.CommandText =

    "INSERT INTO SALES(stor_id,
        ord_num,
        ord_date,
        qty,
        payterms,
        title_id) " +
    "VALUES(@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id)";
```

2. Invoke the **Prepare** method on the **Command** object to prepare the SQL statement for execution. This notifies the database layer that the **Command** object will be executed with bound parameters.

```
statement.Prepare( );
```

3. Declare the parameter objects using the parameter type that matches the **Command** object: **OdbcParameter**, **OleDbParameter**, or **SqlParameter**. After declaring the parameters, create parameter objects by invoking the **Add** method of the **Command** object's **Parameter** collection and assign the return value to the **Parameter** object. The first argument to the **Add** method is the name of the placeholder that the parameter will map to during execution. In the example below, the **stor_id Parameter** object will map to the first item in the **VALUES** clause, which has the placeholder named **@stor_id**. The second argument in the **Add** method is the column type that the placeholder maps to on the server. Look to [Table 5-8](#) for a list of frequently used types.

```
{Odbc|OleDb|Sql}Parameter stor_id, ord_num, ord_date,  
    qty, payterms, title_id;  
  
stor_id = statement.Parameters.Add( "@stor_id", DbType.String );  
ord_num = statement.Parameters.Add( "@ord_num", DbType.String );  
ord_date = statement.Parameters.Add( "@ord_date", DbType.DateTime);  
qty = statement.Parameters.Add( "@qty", DbType.Int16 );  
payterms = statement.Parameters.Add( "@payterms", DbType.String );  
title_id = statement.Parameters.Add( "@title_id", DbType.String );
```

4. In this example, the `Parameter` objects are assigned a value by the user-defined `GetNextSale` function call, which could be implemented like this:

```
static bool GetNextSale(SqlParameter stor_id,  
    SqlParameter ord_num,  
    SqlParameter ord_date,  
    SqlParameter qty,  
    SqlParameter payterms,  
    SqlParameter title_id)  
{  
    // Omitted is the code that would  
    // Read a sale record from a file, or user input, etc.  
  
    // If there are no more sale records, return false.  
    if( !more_records ) return false;  
  
    // Assign values to the parameter objects  
    stor_id.Value = 1234;  
    ord_num.Value = "ABCD.123";  
    ord_date.Value = new DateTime(2003,2,24);  
    qty.Value = 50;  
    payterms.Value = "Net 60";  
    title_id.Value = "SD2043";  
    return true;  
}
```

Notice that the parameters are assigned a value by assigning directly to the `Value` property on the `Parameter` object. The `stor_id` parameter object is assigned a value of 1234, whereas `ord_date` is assigned a C# ADO.NET `DateTime` object. The function returns false if there are no more sales to insert into the table; otherwise the function returns true.

Combined with a `while` loop, the program will continue to insert new sales into the database until the `GetNextSale` function runs out of new records to process.

```
while(GetNextSale(stor_id, ord_num, ord_date, qty, payterms, title_id) )  
{
```

5. Invoking the `ExecuteNonQuery` method on the `Command` object executes the `INSERT` statement, with the bound parameter values replaced for their corresponding placeholders. The `ExecuteNonQuery` method returns the number of rows affected, which will be 1 in this case on a successful single-row insert. This return value is used in error handling and the application will exit if the statement should ever fail to execute.

```
// Execute the statement  
  
int result = statement.ExecuteNonQuery( );  
  
if( result != 1 )  
{  
    // If result isn't 1, then the insert failed.  
    System.Console.WriteLine( "The INSERT failed." );  
    break;  
}
```

Table 5-8. Frequently used parameter object types

DbType object type	Description
<code>AnsiString</code>	Variable-length, non-Unicode character string between 1 and 8,000 characters.
<code>AnsiStringFixedLength</code>	Fixed-length, non-Unicode character string.
<code>Binary</code>	Variable-length binary string between 1 and 8,000 bytes.
<code>Boolean</code>	Represents a true or false value.
<code>Byte</code>	Unsigned integer value ranging 0 to 255.
<code>Currency</code>	Currency value ranging from -2^{63} to $2^{63}-1$.
<code>DateTime</code>	Date and time value.
<code>Decimal</code>	Numeric value ranging from 10^{-28} to $7.9 \cdot 10^{28}$ with about 28 significant digits.
<code>Double</code>	Floating point type ranging from $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$ with about 15 digits.
<code>Int{16,32,64}</code>	Signed integer; the number suffix indicates the number of bits of precision. For example, <code>Int32</code> is a 32-bit integer.
<code>Single</code>	Floating point type ranging from $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$, with about 15 digits.
<code>String</code>	Variable-length, Unicode character string.
<code>StringFixedLength</code>	Fixed-length, Unicode character string.
<code>UInt{16,32,64}</code>	Unsigned integer; the number suffix indicates the number of bits of precision. For example, <code>UInt32</code> is a 32-bit unsigned integer.

5.7.2 Binding Parameters with JDBC

The following Java code fragment executes a SQL `INSERT` statement that adds new sales to the `sales` table in the `pubs` database. The `INSERT` statement is parameterized to provide better performance.

```
// Create a Command object for the SQL statement  
  
PreparedStatement statement = connection.prepareStatement(
```

```
"INSERT INTO SALES(stor_id,
                    ord_num,
                    ord_date,
                    qty,
                    payterms,
                    title_id) " +
"VALUES(?, ?, ?, ?, ?, ?)";

while( getNextSale(statement) )
{
    // Execute the statement
    int result = statement.executeUpdate( );
    if( result != 1 )
    {
        // If result isn't 1, then the insert failed.
        System.out.println( "The INSERT failed." );
        break;
    }
}
```

5.7.2.1 Use the following steps to execute statements with bound parameters in JDBC:

1. Create a JDBC `PreparedStatement` object and pass the parameterized SQL statement into its constructor. The difference for using bound parameters within the statement is in the `VALUES` clause of the `INSERT` statement. Contained within the `VALUES` clause are six placeholders (the question marks) for the parameters that will later be bound to the `PreparedStatement` object.

```
PreparedStatement statement = connection.prepareStatement(
    "INSERT INTO SALES(stor_id,
                      ord_num,
                      ord_date,
                      qty,
                      payterms,
                      title_id) " +
    "VALUES(?, ?, ?, ?, ?, ?)");
```

1. In this example, the parameters are assigned a value by the user-defined `getNextSale` function call, which could be implemented like this:

```
static boolean getNextSale( PreparedStatement statement )
```

```
throws SQLException
{
    // Omitted is the code that would
    // Read a sale record from a file, or user input, etc.

    // If there are no more sale records, return false.
    if( !more_records ) return false;

    statement.setString(1, "1234");
    statement.setString(2, "ABCD.123");
    statement.setDate(3, new java.sql.Date(2003, 2, 24));
    statement.setInt(4,50);
    statement.setString(5, "Net 60");
    statement.setString(6, "SD2043");

    return true;
}
```

1. Each binding position is referenced by its ordinal position in the SQL statement, with the first position starting at 1. The values are assigned to placeholders using the `set` methods found on the `PreparedStatement` object. [Table 5-9](#) contains a list of frequently used `set` methods.
2. The function returns false if there are no more sales to insert into the table; otherwise the function returns true.
3. Combined with a `while` loop, the program will continue to insert new sales into the database until the `getNextSale` function runs out of new records to process.

```
while( getNextSale(statement) )
```

```
{
```

1. Invoking the `executeUpdate` method on the `PreparedStatement` object executes the `INSERT` statement, with the bound parameter values replaced for their corresponding placeholders. The `executeUpdate` method returns the number of rows affected, which will be 1 in this case on a successful single-row insert. This return value is used in error handling and the application will exit if the statement should ever fail to execute.

```
// Execute the statement

int result = statement.executeUpdate( );

if( result != 1 )
{
    // If result isn't 1, then the insert failed.

    System.out.println( "The INSERT failed." );

    break;
}
}
```

Table 5-9. Frequently used PreparedStatement set methods

Method name	Description
-------------	-------------

<code>setBlob(int i, Blob value)</code>	Sets the placeholder at position <i>i</i> to the Blob contained in <i>value</i> .
<code>setBoolean(int i, boolean value)</code>	Sets the placeholder at position <i>i</i> to the boolean contained in <i>value</i> .
<code>setByte(int i, byte value)</code>	Sets the placeholder at position <i>i</i> to the byte contained in <i>value</i> .
<code>setClob(int i, Clob value)</code>	Sets the placeholder at position <i>i</i> to the Clob contained in <i>value</i> .
<code>setDate(int i, Date value[, Calendar cal])</code>	Sets the placeholder at position <i>i</i> to the Date contained in <i>value</i> . If <i>cal</i> is provided, it'll be used to interpret the Date value .
<code>setDouble(int i, double value)</code>	Sets the placeholder at position <i>i</i> to the double contained in <i>value</i> .
<code>setFloat(int i, float value)</code>	Sets the placeholder at position <i>i</i> to the float contained in <i>value</i> .
<code>setInt(int i, int value)</code>	Sets the placeholder at position <i>i</i> to the int contained in <i>value</i> .
<code>setLong(int i, long value)</code>	Sets the placeholder at position <i>i</i> to the long contained in <i>value</i> .
<code>setNull(int i, int v)</code>	Sets the placeholder at position <i>i</i> to a NULL value when <i>v</i> is true.
<code>setString(int i, String value)</code>	Sets the placeholder at position <i>i</i> to the String contained in <i>value</i> .
<code>setTimestamp(int i, Timestamp value[,Calendar cal])</code>	Sets the placeholder at position <i>i</i> to the Timestamp contained in <i>value</i> . If <i>cal</i> is provided, it'll be used to interpret the Timestamp value .

5.8 Error Handling

With everything that can go wrong in a database application, the correct handling of errors is a topic large enough for its own chapter (or book!). We're lucky that the APIs covered in this book, built on the modern Java language and on the .NET framework, contain error handling mechanisms that make recovering from errors easy.

5.8.1 Error Handling in ADO.NET

ADO.NET's error handling model is based on .NET exceptions. Functions or methods that can fail because of an error must be wrapped in a `try/catch/finally` block. The difficult part of handling exceptions is determining what to do when an error occurs and freeing resources for objects that were created. The following code example shows the error handling pattern that can be successfully used with ADO.NET to clean up all resources in the event of an error:

```
{Odbc|OleDb|Sql}Connection connection = null;

{Odbc|OleDb|Sql}Command statement = null;

{Odbc|OleDb|Sql}DataReader resultSet = null;

try {

    // Create and use the Connection, Command, and DataReader objects

    connection = new SqlConnection(connection_string);

    connection.Open( );

    statement = connection.CreateCommand( );

    statement.CommandText = SQL;

    resultSet = statement.ExecuteReader( );

} catch( Exception e ) {

    // Notify user that an error occurred

} finally {

    if( resultSet != null ) resultSet.Close( );

    if( statement != null ) statement.Dispose( );

    if( connection != null ) connection.Close( );

}
```

If an error occurs in the `try` block and an exception (of type `Exception` in this case) is thrown, the exception will be caught and the code in the `catch` block will be executed. The `finally` block, which will be executed regardless of whether an exception is thrown, should be responsible for freeing the resources. Which resources to free is difficult to determine if you don't know the point of failure. For example, if the failure occurs when executing a SQL statement, then the only resources that need to be freed are the `statement` and `connection` objects. In the code fragment above, an exception thrown in the `ExecuteReader` method would prevent the `resultSet` object from being closed, since it would still be set to its initial value of `null`.



It is good programming practice to free resources in the opposite order in which they were allocated.

5.8.2 Error Handling in JDBC

JDBC's error handling model is based on Java exceptions; therefore, the standard **try/catch/finally** error handling paradigm of Java can be applied to JDBC applications. The following Java code fragment is an example of how to use a **try/catch/finally** code block to gracefully handle application exceptions and free allocated resources:

```
Connection connection = null;

Statement statement = null;

ResultSet resultSet = null;

try {

    // Create and use the Connection, Statement, and ResultSet objects

    connection = DriverManager.getConnection( connection_string );

    statement = connection.createStatement( );

    resultSet = statement.executeQuery( SQL );

} catch( Exception e ) {

    // Then, notify the user of an error.

} finally {

    if( resultSet != null )

        try{resultSet.close( );} catch(Exception e) {}

    if( statement != null )

        try{statement.close( );} catch(Exception e) {}

    if( connection != null )

        try{connection.close( );} catch(Exception e) {}

}
```

If an error occurs in the **try** block and an exception (of type **Exception** in this case) is thrown, the exception will be caught and the code in the **catch** block will be executed. The **finally** block, which will be executed regardless of whether an exception is thrown, should be responsible for freeing the resources held by the **connection**, **result set**, and **statement** objects. Note that each object is first tested for a null value before freeing the resources held by the object. This NULL test is to make the code applicable to all error conditions — the code functions if the exception occurs while opening the connection as well as processing the **result set** data. The **close** methods on the JDBC objects can also generate exceptions when freeing resources, so they must be wrapped in their own **try/catch** blocks. This makes coding awkward, but even so, it will be relatively simple if you follow the pattern given here.



It is good programming practice to free resources in the opposite order from what they were allocated.

5.9 Examples

In the previous sections of this chapter, we examined the fundamental components of SQL statement processing found in most database programs, but didn't include any programs in their entirety. In the following section, we combine the necessary steps into a small program that executes a simple *SELECT* statement and prints out the results. The same example is provided for both of the APIs covered in this chapter.

The examples execute a *SELECT* statement and print out the results after establishing connectivity with the database. The *SELECT* statement is as follows:

```
SELECT a.au_lname, a.au_fname, SUM(t.ytd_sales)
FROM authors a, titleauthor, titles t
WHERE titleauthor.au_id = a.au_id and
      titleauthor.title_id = t.title_id
GROUP BY a.au_lname, a.au_fname
ORDER BY 3 DESC
```

The examples execute the statement and print out the three-column result set returned by the database server.

5.9.1 ADO.NET Example

The following C# ADO.NET database program connects to a database and prints a list of authors in the **pubs** database that includes their year-to-date sales. This program can be easily adapted to meet other database processing needs by following the practices outlined in the earlier sections of this chapter.

```
using System;
using System.Data.SqlClient;

class ExampleApplication
{
    static void Main(string[] args)
    {
        String connection_string =
            "Server=(local);Trusted_Connection=true;DATABASE=pubs;";
        String SQL =
            "SELECT a.au_lname, a.au_fname, SUM(t.ytd_sales) " +
            "FROM authors a, titleauthor, titles t " +
            "WHERE titleauthor.au_id = a.au_id and " +
            "      titleauthor.title_id = t.title_id " +
            "GROUP BY a.au_lname, a.au_fname " +
            "ORDER BY 3 DESC";

        SqlConnection connection = null;
        SqlCommand statement = null;
        SqlDataReader resultSet = null;
```

```
try
{
    // Create Connection and Connect to the Server
    connection = new SqlConnection(connection_string);
    connection.Open( );

    // Create a Command object for the SQL statement
    statement = connection.CreateCommand( );
    statement.CommandText = SQL;

    // Create a Reader for reading the result set
    resultSet = statement.ExecuteReader( );
    while( resultSet.Read( ) )
    {
        // Extract the data from the server and display it
        String fname = "NULL";
        String lname = "NULL";
        String sales = "ZERO";
        if( !resultSet.IsDBNull( 0 ) )
            fname = resultSet.GetString( 0 );
        if( !resultSet.IsDBNull( 1 ) )
            lname = resultSet.GetString( 1 );
        if( !resultSet.IsDBNull( 2 ) )
            sales = resultSet.GetInt32( 2 ).ToString( );
        System.Console.WriteLine( lname + ", " +
            fname + " has sales of " +
            sales);
    }
}

catch( SqlException e )
{
    // Print out the error string, if any.
    System.Console.WriteLine("Error:" + e.ToString( ) );
} finally {
    // Free up resources
    if( resultSet != null ) resultSet.Close( );
    if( statement != null ) statement.Dispose( );
}
```

```
        if( connection != null ) connection.Close( );  
    }  
}  
}
```

5.9.2 JDBC Example

The following Java JDBC database program connects to a database and prints out each of the authors in the **pubs** database, as well as their year-to-date sales.

```
import java.sql.*;  
  
public class ExampleApplication  
{  
    public static void main(String[] args)  
    {  
        String connection_string =  
            "jdbc:microsoft:sqlserver://localhost:1433;" +  
            "User=montoyaj;Password=12345;DatabaseName=pubs;";  
  
        String SQL =  
            "SELECT a.au_lname, a.au_fname, SUM(t.ytd_sales) " +  
            "FROM authors a, titleauthor, titles t " +  
            "WHERE titleauthor.au_id = a.au_id and " +  
            "    titleauthor.title_id = t.title_id " +  
            "GROUP BY a.au_lname, a.au_fname " +  
            "ORDER BY 3 DESC";  
  
        Connection connection = null;  
        Statement statement = null;  
        ResultSet resultSet = null;  
  
        try  
        {  
            Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");  
  
            // Create Connection and Connect to the Server  
            connection = DriverManager.getConnection( connection_string );  
  
            // Create a Command object for the SQL statement
```

```
statement = connection.createStatement( );

// Create a Reader for reading the result set
resultSet = statement.executeQuery( SQL );
while( resultSet.next( ) )
{
    // Extract the data from the server and display it
    String lname = resultSet.getString( 1 );
    if( resultSet.isNull( ) ) lname = "NULL";
    String fname = resultSet.getString( 2 );
    if( resultSet.isNull( ) ) fname = "NULL";
    String sales = resultSet.getString( 3 );
    if( resultSet.isNull( ) ) sales = "ZERO";
    System.out.println( lname + ", " +
        fname + " has sales of " + sales);
}

}

catch( Exception e )
{
    // Print out the error string, if any.
    System.out.println("Error:" + e.toString( ) );
} finally {
    // Free up resources
    if( resultSet != null )
        try {resultSet.close( );} catch( Exception e ) {}
    if( statement != null )
        try {statement.close( );} catch( Exception e ) {}
    if( connection != null )
        try {connection.close( );} catch( Exception e ) {}
}

}

}
```



Appendix A. Sybase Adaptive Server

For the most part, Microsoft SQL Server's dialect of SQL documented throughout this book applies to Sybase Adaptive Server Enterprise 12.5. This appendix documents those datatypes, statements, functions, and keywords that differ between these two popular platforms.



A.1 Sybase Adaptive Server Datatypes

[Table A-1](#) compares the datatypes supported by Sybase Adaptive Server to those supported by Microsoft SQL Server. While there is significant overlap in the types offered, a few types differ in meaning or are missing on one of the platforms. Where differences in the types exist, they'll be listed in the table. Types that are not listed within the table should be considered identical on both platforms.

Table A-1. Sybase Adaptive Server datatypes

Sybase adaptive server datatype	Microsoft SQL Server datatype	Notes
N/A	<i>BIGINT</i>	Usage of <i>decimal(19,0)</i> as an equivalent to <i>BIGINT</i> will allow storage of values within the same range.
<i>BINARY</i>	<i>BINARY</i>	Describes a fixed-length binary value up to 255 bytes in size. In contrast, Microsoft SQL Server can store 8,000 bytes.
<i>CHAR(N)</i>	<i>CHAR(N)</i>	Holds fixed-length character data up to the database's page size in length.
<i>DATETIME</i>	<i>DATETIME</i>	Holds date and time data within the range of 1753-01-01 00:00:00 through 9999-12-31 23:59:59. The <i>DATETIME</i> type can store milliseconds with a granularity of 1/300 second.
<i>NCHAR(N)</i>	N/A	Holds fixed-length character strings in the national character set up to the database's page size in length. In contrast, Microsoft SQL Server's <i>NCHAR</i> datatype stores Unicode character strings.
N/A	<i>NTEXT</i>	Holds Unicode text passages up to 1,073,741,823 characters in length.
<i>NVARCHAR(N)</i>	N/A	Holds variable-length character strings in the national character set up to the database's page size in length. In contrast, Microsoft SQL Server's <i>NVARCHAR</i> datatype stores Unicode character strings.
<i>TIMESTAMP</i>	<i>ROWVERSION</i>	A unique number within a database that is updated whenever a row is updated.
<i>SMALLDATETIME</i>	<i>SMALLDATETIME</i>	Holds date and time data within the range of 1900-01-01 00:00 through 2079-12-31 23:59. Unlike Microsoft SQL Server, Sybase cannot hold dates with second resolution in a <i>SMALLDATETIME</i> .
N/A	<i>SQL_VARIANT</i>	Sybase has no weakly typed storage type like SQL Server's <i>SQL_VARIANT</i> .
<i>UNICHAR</i>	<i>NCHAR(N)</i>	Stores fixed-length, Unicode character strings up to the database's page size in length.
N/A	<i>UNIQUEIDENTIFIER</i>	Represents a value that is unique across all databases and all servers.
<i>UNIVARCHAR</i>	<i>NCHAR(N)</i>	Stores variable-length, Unicode character strings up to the database's page size in length.
<i>VARBINARY(N)</i>	<i>VARBINARY(N)</i>	Describes a variable-length binary value up to 255 bytes in size. In contrast, Microsoft SQL Server can store up to 8,000 bytes.
<i>VARCHAR(N)</i>	<i>VARCHAR(N)</i>	Holds fixed-length character data up to the database's page size in length.

The primary differences between the types offered by SQL Server and Sybase are the storage sizes, precision, and default meaning of the *NCHAR* type. When converting data from one platform to the other, it is important to observe these subtle differences to avoid data truncation or corruption.

A.2 Sybase Adaptive Server SQL Statements

As we saw in the previous section, there is significant overlap in the type support offered by Sybase and SQL Server. This section compares the SQL dialects of Sybase to SQL Server. In most cases, the SQL Server statements described in [Chapter 3](#) apply to Sybase as well. This section outlines those statements that are different between the two platforms.

In [Table A-2](#), the SQL commands are listed along with notes containing the differences between SQL Server and Sybase. While most of the statements have identical syntax and meaning, for those statements that contain large differences, the syntax for Sybase is listed with notes on how the statements differ.

Table A-2. Sybase Adaptive Server SQL statements

Command	Notes
<i>ALTER PROCEDURE</i>	Sybase does not support the <i>ENCRYPTION</i> keyword or replication through the <i>FOR REPLICATION</i> keywords.
<i>ALTER TABLE</i>	<p>Sybase does not support the <i>ROWGUIDCOL</i> keyword or the <i>CHECK</i> and <i>NOCHECK</i> keywords for enabling constraints.</p> <p><i>ALTER TABLE table_name</i></p> <p>[<i>ADD {column_name datatype attributes}{,...}</i>]</p> <p> [<i>DROP column_name{,...}</i>]</p> <p> [<i>ADD CONSTRAINT {constraint_name constraint_clause}{,...}</i>]</p> <p> [<i>DROP CONSTRAINT constraint_name{,...}</i>]</p> <p> [<i>MODIFY {column_name data_type [[NOT] NULL]}{,...}</i>]</p> <p> [<i>{ENABLE DISABLE} trigger_name</i>]</p>
<i>ALTER TRIGGER</i>	Sybase has no equivalent to the <i>ALTER TRIGGER</i> statement in SQL Server. For equivalent functionality, drop the existing trigger with a <i>DROP TRIGGER</i> statement and recreate it with the <i>CREATE TRIGGER</i> statement.
<i>ALTER VIEW</i>	Sybase has no equivalent to the <i>ALTER VIEW</i> statement in SQL Server. For equivalent functionality, drop the existing view with a <i>DROP VIEW</i> statement and recreate it with the <i>CREATE VIEW</i> statement.
<i>CALL</i>	Not supported by SQL Server or Sybase. Look to the <i>EXECUTE</i> command for this functionality.
<i>CONNECT</i>	<p>Supported with variations.</p> <p><i>CONNECT TO server_name</i></p>
<i>CREATE DATABASE</i>	<p>Creating a new database in Sybase has the following syntax:</p> <p><i>CREATE DATABASE database_name</i></p> <p>[<i>ON { DEFAULT database_device} [=size]</i></p> <p> <i>{ database_device [=size]}{,...}</i>]</p> <p>[<i>LOG ON {log_device [=size]}{,...}</i>]</p> <p>[<i>WITH {OVERRIDE DEFAULT_LOCATION = "pathname"}</i>]</p> <p>[<i>FOR {LOAD PROXY_UPDATE}</i>]</p> <ul style="list-style-type: none"> • <i>DEFAULT</i> specifies the default device, equivalent to the <i>PRIMARY</i> keyword on SQL Server. • <i>size</i> is the size of the database or transaction log in megabytes. • <i>log_device</i> has the same meaning as SQL Server.

	<ul style="list-style-type: none"> • <i>PROXY_UPDATE</i> automatically gets the metadata from the <i>DEFAULT_LOCATION</i> when creating the database.
<i>CREATE FUNCTION</i>	Support is identical to that offered by SQL Server, except that the following options are unsupported: <i>VARYING</i> , <i>ENCRYPTION</i> , and <i>FOR REPLICATION</i> .
<i>CREATE INDEX</i>	<p>Differences exist within the options provided in the <i>WITH</i> clause, as follows:</p> <pre>[WITH [{FILLFACTOR = <i>fillfactor</i> MAX_ROWS_PER_PAGE = <i>num_rows</i>}] [[,]RESERVEPAGEGAP = <i>num_pages</i>] [[,]CONSUMERS = X] [[,]IGNORE_DUP_KEY] [[,]IGNORE_DUP_ROW ALLOW_DUP_ROW] [[,]SORTED_DATA] [[,]STATISTICS USING <i>num_steps</i> VALUES]</pre>
	<ul style="list-style-type: none"> • <i>FILLFACTOR</i> has the same meaning as SQL Server. • <i>MAX_ROWS_PER_PAGE</i> is another way of controlling the fill factor of the index, but uses the row count instead of a percentage of the page space. • <i>RESERVEPAGEGAP</i> provides a method of controlling the ratio of empty index pages to filled ones. This can provide a performance advantage for indexes that grow in size frequently. Valid values for <i>num_pages</i> are 0-255, with the default being 0. • <i>IGNORE_DUP_KEY</i> has the same meaning as SQL Server.
	<ul style="list-style-type: none"> • <i>IGNORE_DUP_ROW</i> allows the creation of a clustered index on tables that contain duplicate rows. While the index can be created when the table contains duplicate rows, any insert or update statement that can result in the creation of a duplicate row will be canceled once the index is created.
<i>CREATE INDEX</i>	<ul style="list-style-type: none"> • <i>ALLOW_DUP_ROW</i> is similar to <i>IGNORE_DUP_ROW</i>, except that it permits insert and update statements that will result in duplicate rows. • <i>SORTED_DATA</i> speeds up the creation of indexes for tables that are already sorted on disk. • <i>WITH STATISTICS USING num_steps VALUES</i> controls the amount of statistics maintained and provided to the query optimizer.
<i>CREATE PROCEDURE</i>	Support is identical to that offered by SQL Server, except that the following options are unsupported: <i>VARYING</i> , <i>ENCRYPTION</i> , and <i>FOR REPLICATION</i> .
<i>CREATE ROLE</i>	<p>Supported with variations.</p> <pre>CREATE ROLE <i>role_name</i> [WITH PASSWD "<i>password</i>" [, {"PASSWD EXPIRATION" "MIN PASSWD LENGTH" "MAX FAILED LOGINS" } <i>option_value</i>][,...]]</pre> <p>This statement will create a role by <i>role_name</i> with an optional password. Unique to Sybase is the ability to specify simple security options for the new role.</p>
	<p>While casual users of SQL Server and Sybase will notice no differences in the CREATE TABLE syntax, others will note the following differences:</p> <ul style="list-style-type: none"> • The NOT FOR REPLICATION option is not supported on Sybase. • The ROWGUIDCOL column attribute is not supported on Sybase.

- The TEXTIMAGE_ON option is not supported on Sybase.
- The IDENTITY attribute cannot have a seed or increment value specified on Sybase.

Other than those small changes, the features unique to Sybase are listed below.

```
CREATE TABLE [database_name.owner].table_name
  ({column_name datatype
  {[DEFAULT default_value]
  | [IDENTITY | NULL | NOT NULL]
  | [OFF ROW | IN ROW [ (size_in_bytes) ] ]
  | REFERENCES [[database_name.]owner.]ref_table [(ref_column)]
  {UNIQUE | PRIMARY KEY} [CLUSTERED | NONCLUSTERED] [asc |
  desc]
  [WITH { FILLFACTOR = pct,
  MAX_ROWS_PER_PAGE = num_rows, }
  RESERVEPAGEGAP = num_pages }
  [ON segment_name]
  | CHECK (search_condition)
  }
  [[CONSTRAINT constraint_name]
  | FOREIGN KEY ({column_name}[,...]) REFERENCES
  [[database_name.]owner.]ref_table
  [[ref_column][,...]]
  | CHECK (search_condition)
  {UNIQUE | PRIMARY KEY} [CLUSTERED | NONCLUSTERED]
  ({column_name [ASC | DESC]}[,...])
  [WITH { FILLFACTOR = pct,
  MAX_ROWS_PER_PAGE = num_rows,
  RESERVEPAGEGAP = num_pages } ]
  [ON segment_name][,...])
  [LOCK {DATAROWS | DATAPAGES | ALLPAGES }]
  [WITH { MAX_ROWS_PER_PAGE = num_rows,
  EXP_ROW_SIZE = num_bytes,
  RESERVEPAGEGAP = num_pages,
  IDENTITY_GAP = value }]
  [ON segment_name ]
  [ [ EXTERNAL TABLE ] AT pathname ]
```

CREATE
TABLE

- OFF/IN ROW specifies if a Java-SQL column is physically stored inside a row or outside. The SIZE_IN_BYTES option is the maximum space required to store an IN ROW type.
- ASC/DESC determines the ordering for an index created for a constraint. Ascending order,

CREATE TABLE	<p>ASC, is the default.</p> <ul style="list-style-type: none"> • <i>MAX_ROWS_PER_PAGE</i> limits the number of rows per page. • <i>LOCK</i> specifies the locking strategy used for the table. • <i>EXP_ROW_SIZE</i> specifies the expected row size in bytes. The default is zero, which means the server's default will be used. • <i>RESERVEPAGEGAP</i> specifies the desired ratio of filled to empty pages. Valid values are 0-255, with a default of zero. • <i>IDENTITY_GAP</i> controls the gap between consecutive values in identity columns. • <i>EXTERNAL TABLE</i> specifies that the table is stored externally. This is the default, so its usage is optional.
CREATE TRIGGER	<p>Sybase offers support similar to SQL Server, with the following exceptions:</p> <ul style="list-style-type: none"> • <i>WITH ENCRYPTION</i> is not permitted. • <i>AFTER</i> and <i>INSTEAD OF</i> are not permitted. • <i>WITH APPEND</i> is not permitted. • <i>NOT FOR REPLICATION</i> is not permitted. • <i>COLUMNS_UPDATED()</i> function is not supported.
CREATE VIEW	<p>Support is identical to that offered by SQL Server except that Sybase has no <i>ENCRYPTION</i>, <i>SCHEMABINDING</i>, or <i>VIEW_METADATA</i> options.</p>
DECLARE CURSOR	<p>Support is identical to that offered by SQL Server, except that Sybase has no <i>INSENSITIVE</i> or <i>SCROLL</i> options.</p>
DELETE	<p>The <i>DELETE</i> statement is nearly identical between Sybase and SQL Server. SQL Server supports a <i>WITH</i> and <i>OPTION</i> clause that Sybase does not. The features unique to Sybase are listed below.</p> <pre>DELETE [[owner.]{table_name view_name}] [FROM {[owner.] {view_name [READPAST] table_name [READPAST] [(INDEX {index_name table_name} [PREFETCH size][LRU MRU])]}],...] [WHERE {search_conditions CURRENT OF cursor_name}]] [PLAN "abstract_plan"]</pre> <p><i>READPAST</i> instructs the server to skip over all pages or rows currently locked by other transactions, deleting rows only from pages not currently in use.</p> <p><i>PREFETCH</i> specifies the I/O size, in kilobytes, for tables that are bound to caches.</p> <p><i>LRU/MRU</i> specifies either a least or most recently used buffer replacement strategy.</p> <p><i>PLAN</i> sends <i>abstract_plan</i> as an alternative execution plan to the query optimizer.</p>
DISCONNECT	<p>Supported with the following command:</p> <pre>DISCONNECT</pre>
	<p>Supported with the following syntax:</p> <pre>DROP ROLE role_name [WITH OVERRIDE]</pre>

<i>DROP ROLE</i>	Using <i>WITH OVERRIDE</i> will ignore all restrictions on dropping the role from the databases.
<i>FETCH</i>	<p>The <i>FETCH</i> command differs significantly from SQL Server. Sybase has sequential server-side cursors, so the <i>NEXT</i>, <i>PRIOR</i>, <i>FIRST</i>, and <i>LAST</i> options are not permitted.</p> <p>Additionally, parameters can be used in a target list as long as there is a one-to-one mapping between these items and the ones returned by the <i>SELECT</i> statement when the cursor was created.</p> <p><i>FETCH cursor_name [INTO fetch_target_list]</i></p> <p>For example:</p> <pre>DECLARE authors_cursor CURSOR FOR SELECT au_lname, au_fname FROM authors OPEN authors_cursor FETCH authors_cursor INTO @lname, @fname GO</pre>
<i>GRANT</i>	Support is identical to that offered by SQL Server, except that Sybase has no <i>AS</i> clause.
<i>INSERT</i>	Sybase offers the SQL99 syntax for <i>INSERT</i> statements; therefore, Sybase permits none of the extended features of SQL Server.
<i>RETURN</i>	Sybase's return statement <i>RETURN</i> , unlike SQL Server, cannot return NULL values.
<i>REVOKE</i>	<p>Sybase offers support similar to SQL Server with the following exceptions:</p> <ul style="list-style-type: none"> • The <i>TO</i> clause cannot be used in place of <i>FROM</i>. • The <i>AS</i> clause is not permitted.
<i>SAVEPOINT</i>	Support identical to SQL Server's <i>SAVE TRANSACTION</i> statement.
<i>SELECT</i>	<p>Sybase offers support for <i>SELECT</i> statements identical to SQL Server, with the following exceptions:</p> <ul style="list-style-type: none"> • Sybase has a <i>PLAN</i> clause instead of an <i>OPTION</i> clause used for passing hints to the server's statement optimizer. • Sybase has no <i>TOP</i> clause. • Sybase does not support <i>WITH CUBE</i> or <i>WITH ROLLUP</i> in the <i>GROUP BY</i> clause. • Sybase offers a <i>FOR { UPDATE READ ONLY }</i> clause that can only be used with a stored procedure when the query defines the result for a cursor. • Sybase has <i>AT ISOLATION { 0, 1, 2, 3 }</i> for choosing a non-default isolation level for query execution. • Sybase also offers a <i>SELECT INTO</i> statement, but with an additional, optional clause for controlling the locking: <i>LOCK { DATAROWS DATAPAGES ALLPAGES }</i>
<i>SET ROLE</i>	<p>Sybase supports the <i>SET ROLE</i> statement with the following syntax:</p> <pre>SET ROLE {"SA_ROLE" "SSO_ROLE" "OPER_ROLE" role_name [WITH PASSWD "password"]} { ON OFF }</pre>
<i>SET TIME ZONE</i>	The <i>SET TIMEZONE</i> statement is not supported in Sybase or SQL Server.
<i>START TRANSACTION</i>	Support is offered through <i>BEGIN TRANSACTION</i> , which is identical to SQL Server, except that parameterized transaction names are not supported.
	The <i>UPDATE</i> statement is nearly identical between Sybase and SQL Server. SQL Server supports

UPDATE	<p>WITH and OPTION clauses that Sybase does not. The features unique to Sybase are listed below.</p> <pre>UPDATE {table_name view_name} SET {[table_name.view_name.}] column_name1 = {expression1 NULL (select_statement)} variable_name1 = {expression1 NULL (select_statement)} [,...] [FROM {view_name [READPAST]] table_name [READPAST] [(INDEX {index_name table_name } [PREFETCH size][LRU MRU])][,...] [WHERE {search_conditions CURRENT OF cursor_name}] [PLAN "abstract_plan"]</pre>
UPDATE	<ul style="list-style-type: none">• READPAST instructs the server to skip over all pages or rows currently locked by other transactions, updating rows only from pages not currently in use.• PREFETCH specifies the I/O size, in kilobytes, for tables that are bound to caches.• LRU/MRU specifies either the least or most recently used buffer replacement strategy.• PLAN sends abstract_plan as an alternative execution plan to the query optimizer.

A.3 Sybase Adaptive Server SQL Functions

This section compares the functions provided by Sybase and SQL Server. [Table A-3](#) provides a list of the functions support differ on SQL Server. Only functions with differences between the two platforms have been listed.

Table A-3. Sybase Adaptive Server-supported function

Sybase function	SQL Server function	
To retrieve the application name on Sybase, use the following SQL statement: <pre>select program_name from master..sysprocesses where spid = @@spid</pre>	<i>APP_NAME()</i>	Retu by a
N/A	<i>BINARY_CHECKSUM({* expression [,...]})</i>	Retu row
N/A	<i>CAST(expression AS datatype)</i>	Conv spec Ente
<i>CHAR_LENGTH(expression)</i>	N/A	Retu expr prov
N/A	<i>CHECKSUM({* expression [,...]})</i>	Retu valu
N/A	<i>CHECKSUM_AGG([ALL DISTINCT] expression)</i>	Retu
N/A	<i>COALESCE(expression [,...])</i>	Retu argu
<i>COMPARE(char_expression1, char_expression2 [, {collation_name collation_ID}])</i>	N/A	Retu colla 1 : c char. 0: cl -1 : char.
N/A	<i>CONTAINS({column * }, 'contains_search_condition')</i>	Sear cont func to th
N/A	<i>CONTAINSTABLE(table, column, contains_search_condition)</i>	Retu cont func to th
N/A	<i>COUNT_BIG([ALL DISTINCT] expression)</i>	Sam
N/A	<i>DATABASEPROPERTYEX(database, property)</i>	Retu
N/A	<i>DAY(date)</i>	Retu the c
N/A	<i>FILE_ID('file_name')</i>	Retu
N/A	<i>FILE_NAME(file_id)</i>	Retu
N/A	<i>FILEGROUP_ID('filegroup_name')</i>	Retu
N/A	<i>FILEGROUP_NAME(filegroup_id)</i>	Retu
N/A	<i>FILEGROUPPROPERTY(filegroup_name, property)</i>	Retu prop
N/A	<i>FILEPROPERTY(file, property)</i>	Retu prop

N/A	<i>FULLTEXTCATALOGPROPERTY</i> (catalog_name, property)	Retu
N/A	<i>FULLTEXTSERVICEPROPERTY</i> (property)	Retu
N/A	<i>FORMATMESSAGE</i> (msg_number, param_value [,...])	Cons SYS
N/A	<i>FREETEXTTABLE</i> (table { column *}, 'freetext_string' [, top_n_by_rank])	Usec colur matc
N/A	<i>GETANSINULL</i> (['database'])	Retu
N/A	<i>GETUTCDATE</i> ()	Retu
N/A	<i>GROUPING</i> (column_name)	Retu ROLI
<i>HEXTOINT</i> (hexadecimal_string)	N/A	Retu argu CON How hexa with
N/A	<i>IDENT_INCR</i> ('table_or_view')	Retu
N/A	<i>IDENT_SEED</i> ('table_or_view')	Retu
N/A	<i>IDENT_CURRENT</i> ('table_name')	Retu spec
N/A	<i>IDENTITY</i> (data_type [, seed, increment]) As column_name	Usec ideni
N/A	<i>INDEXPROPERTY</i> (table_id, index, property)	Retu
<i>INTTOHEX</i> (integer_expression)	N/A	Retu integ SQL integ
N/A	<i>ISDATE</i> (expression)	Valic DATi
N/A	<i>IS_MEMBER</i> ({'group' 'role'})	Retu whet Serv Syba
N/A	<i>IS_SRVROLEMEMBER</i> ('role' [, 'login'])	Retu whet
N/A	<i>ISNUMERIC</i> (expression)	Valic NUM use i SELE NOT
<i>IS_SEC_SERVICE_ON</i> ('security_service')	N/A	Retu 0 ott
<i>lct_admin</i> ({'LASTCHANCE' 'LOGFULL'})	N/A	Modi proc proc
N/A	<i>LEFT</i> (character_expression, integer_expression)	Retu start
N/A	<i>LEN</i> (string_expression)	Retu expr funci
<i>LICENSE_ENABLED</i> ('feature')	N/A	Retu
N/A.	<i>MONTH</i> (date)	Retu
<i>MUT_EXCL_ROLES</i> (role1, role2 [, membership [, activation]])	N/A	Retu rovl

<i>{membership activation }</i>		Execu
N/A	<i>NCHAR(integer_expression)</i>	Retu integ funci
N/A	<i>NEWID()</i>	Crea UNIC
N/A	<i>NULLIF(expression, expression)</i>	Retu equi
N/A	<i>OBJECTPROPERTY(id, property)</i>	Retu data
N/A	<i>OPEN{[Global]cursor_name cursor_variable_name}</i>	Oper
N/A	<i>OPENDATASOURCE(provider_name, init_string)</i>	Mak linke
N/A	<i>OPENQUERY(linked_server, 'query')</i>	Quer a lin
N/A	<i>OPENROWSET('provider_name', {'datasource'; 'user_id',password 'provider_string'},{[catalog.][schema.]object 'query'})</i>	Quer as a
N/A	<i>PARSENAME('object_name', object_piece)</i>	Retu nam <i>Obje</i>
<i>PATINDEX('%pattern%', expression)</i>	N/A	Retu patt
N/A	<i>PERMISSIONS(object_id, column)</i>	Retu user
<i>PROC_ROLE('role_name')</i>	N/A	Retu the r the j funci
N/A	<i>REPLACE('string_expression1','string_expression2','string_expression3')</i>	Repl with Sam
<i>ROLE_CONTAIN('role1','role2')</i>	N/A	Retu
<i>ROLE_ID(role_name)</i>	N/A	Retu <i>role_</i> state <i>SELE</i> <i>issqlr</i> This role
<i>ROLE_NAME(role_id)</i>	N/A	Retu equ state <i>SELE</i> <i>uid =</i>
<i>ROWCNT(sysindexes.doampg)</i>	N/A	Prov table use f <i>SELE</i> <i>WHE</i>
N/A	<i>ROWCOUNT_BIG()</i>	Retu (sam

N/A	<i>SESSION_USER</i>	Retu
<i>SHOW_ROLE()</i>	N/A	Retu
<i>SHOW_SEC_SERVICES()</i>	N/A	Retu
<i>SORTKEY(char_expr [, {collation_name collation_id}])</i>	N/A	Retu valu
N/A	<i>STATS_DATE(table_id, index_id)</i>	Retu last
N/A	<i>STDEV(expression)</i>	Retu colur
N/A	<i>STDEVP(expression)</i>	Retu valu
<i>SUSER_ID(['login'])</i>	N/A	Retu spec func
N/A	<i>SUSER_SID(['login'])</i>	Retu spec equi
<i>SUSER_NAME([server_user_id])</i>	N/A	Retu spec equi
N/A	<i>SUSER_SNAME([server_user_sid])</i>	Retu spec equi
<i>SYB_SENDMSG(ip_address, port, message)</i>	N/A	Send argu
N/A	<i>SYSTEM_USER</i>	Retu
<i>TO_UNICHAR(integer)</i>	N/A	Retu equi use 1
<i>TSEQUEL(timestamp_expr, timestamp_literal)</i>	N/A	Retu time Serv equi
N/A	<i>TYPEPROPERTY(datatype,property)</i>	Retu
<i>UHIGHSURR(uchar_expr, start)</i>	N/A	Retu posit
<i>ULOWSURR(uchar_expr, start)</i>	N/A	Retu posit
N/A	<i>UNICODE('ncharacter_expression')</i>	Retu char equi
<i>USCALAR(uchar_expr)</i>	N/A	Retu char equi
<i>VALID_NAME(user_name)</i>	N/A	Retu user
<i>VALID_USER(user_id)</i>	N/A	Retu iden
N/A	<i>VAR(expression)</i>	Retu
N/A	<i>VARP(expression)</i>	Retu valu
N/A	<i>YEAR(date)</i>	Retu spec

A.4 Sybase Adaptive Server Keywords

This section lists the keywords used by Sybase Adaptive Server Enterprise.

ABSOLUTE	ACTION	ADD	ALL
ALLOCATE	ALTER	AND	ANY
ARE	ARITH_OVERFLOW	AS	ASC
ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH
BOTH	BREAK	BROWSE	BULK
BY	CASCADE	CASCADED	CASE
CAST	CATALOG	CHAR	CHAR_CONVERT
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CHECKPOINT	CLOSE	CLUSTERED	COALESCE
COLLATE	COLLATION	COLUMN	COMMIT
COMPUTE	CONFIRM	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONTINUE	CONTROLROW
CONVERT	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DATE	DAY	DBCC	DEALLOCATE
DEC	DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE	DESC
DESCRIBE	DESCRIPTOR	DETERMINISTIC	DIAGNOSTICS
DISCONNECT	DISK DISTINCT	DOMAIN	DOUBLE
DROP	DUMMY	DUMP	ELSE
END	END-EXEC	ENDTRAN	ERRLVL
ERRORDATA	ERROREXIT	ESCAPE	EXCEPT
EXCEPTION	EXCLUSIVE	EXEC	EXECUTE
EXISTS	EXIT	EXP_ROW_SIZE	EXTERNAL
EXTRACT	FALSE	FETCH	FILLFACTOR
FIRST	FLOAT	FOR	FOREIGN
FOUND	FROM	FULL	FUNC
FUNCTION	GET	GLOBAL	GO
GOTO	GRANT	GROUP	HAVING
HOLDLOCK	HOUR	IDENTITY	IDENTITY_GAP
IDENTITY_INSERT	IDENTITY_START	IF	IMMEDIATE
IN	INDEX	INDICATOR	INITIALLY
INNER	INOUT	INPUT	INSENSITIVE
INSERT	INSTALL	INT	INTEGER
INTERSECT	INTERVAL	INTO	IS
ISOLATION	JAR	JOIN	KEY
KILL	LANGUAGE	LAST	LEADING

LEFT	LEVEL	LIKE	LINENO
LOAD	LOCAL	LOCK	LOWER
MATCH	MAX	MAX_ROWS_PER_PAGE	MIN
MINUTE	MIRROR	MIRROREXIT	MODIFY
MODULE	MONTH	NAMES	NATIONAL
NATURAL	NCHAR	NEW	NEXT
NO	NOHOLDLOCK	NONCLUSTERED	NOT
NULL	NULLIF	NUMERIC	NUMERIC_TRUNCATION
OCTET_LENGTH	OF	OFF	OFFSETS
ON	ONCE	ONLINE	ONLY
OPEN	OPTION	OR	ORDER
OUT	OUTER	OUTPUT	OVER
OVERLAPS	PAD	PARTIAL	PARTITION
PERM	PERMANENT	PLAN	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY
PRINT	PRIOR	PRIVILEGES	PROC
PROCEDURE	PROCESSEXIT	PROXY_TABLE	PUBLIC
QUIESCE	RAISERROR	READ	READPAST
READTEXT	REAL	RECONFIGURE	REFERENCES REMOVE
RELATIVE	REORG	REPLACE	REPLICATION
RESERVEPAGEGAP	RESTRICT	RETURN	RETURNS
REVOKE	RIGHT	ROLE	ROLLBACK
ROWCOUNT	ROWS	RULE	SAVE
SCHEMA	SCROLL	SECOND	SECTION
SELECT	SESSION_USER	SET	SETUSER
SHARED	SHUTDOWN	SIZE	SMALLINT
SOME	SPACE	SQL	SQLCODE
SQLERROR	SQLSTATE	STATISTICS	STRINGSIZE
STRIPE	SUBSTRING	SUM	SYB_IDENTITY
SYB_RESTREE	SYB_TERMINATE	SYSTEM_USER	TABLE
TEMP	TEMPORARY	TEXTSIZE	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRAN	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER	TRIM
TRUE	TRUNCATE	TSEQUAL	UNION
UNIQUE	UNKNOWN	UNPARTITION	UPDATE
UPPER	USAGE	USE	USER
USER_OPTION	USING	VALUE	VALUES
VARCHAR	VARYING	VIEW	WAITFOR
WHEN	WHENEVER	WHERE	WHILE
WITH	WORK	WRITE	WRITETEXT
YEAR	ZONE		



Appendix B. Shared and Platform-Specific Keywords

The tables below display the keywords in the ANSI standard, and in the five platform-specific implementations of SQL that are discussed in this book, with the Sybase Adaptive Server. The keyword tables appear in the following order:

- [Shared keywords](#)
- [SQL2003 keywords](#)
- [DB2 keywords](#)
- [MySQL keywords](#)
- [Oracle keywords](#)
- [PostgreSQL keywords](#)
- [SQL Server keywords](#)
- [Sybase Adaptive Server keywords](#)

Table B-1. Shared keywords

ADD	ALL	ALTER	AND
AS	ASC	BY	CHECK
COLUMN	CREATE	DATE	DEFAULT
DELETE	DESC	DROP	FOR
FROM	IN	INTO	IS
LIKE	NOT	NULL	ON
OR	ORDER	REVOKE	SELECT
SET	TABLE	THEN	TO
UNIQUE	UPDATE	WITH	

Table B-2. SQL2003 keywords

ABSOLUTE	ACTION	ADD	ADMIN
AFTER	AGGREGATE	ALIAS	ALL
ALLOCATE	ALTER	AND	ANY
ARE	ARRAY	AS	ASC
ASSERTION	ASSERTION	AT	ATOMIC
AUTHORIZATION	BEFORE	BEGIN	BIGINT
BINARY	BIT	BLOB	BOOLEAN
BOTH	BREADTH	BY	CALL
CASCADE	CASCADEDE	CASE	CAST
CATALOG	CHAR	CHARACTER	CHECK
CLASS	CLOB	CLOSE	COLLATE
COLLATION	COLLECT	COLUMN	COMMIT
COMPLETION	CONDITION	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONSTRUCTOR	CONTAINS

CONTINUE	CORRESPONDING	CREATE	CROSS
CUBE	CURRENT	CURRENT_DATE	CURRENT_PATH
CURRENT_ROLE	CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER
CURSOR	CYCLE	DATA	DATALINK
DATE	DAY	DEALLOCATE	DEC
DECIMAL	DECLARE	DEFAULT	DEFERRABLE
DELETE	DEPTH	DEREF	DESC
DESCRIPTOR	DESTRUCTOR	DIAGNOSTICS	DICTIONARY
DISCONNECT	DO	DOMAIN	DOUBLE
DROP	ELEMENT	END-EXEC	EQUALS
ESCAPE	EXCEPT	EXCEPTION	EXECUTE
EXIT	EXPAND	EXPANDING	FALSE
FIRST	FLOAT	FOR	FOREIGN
FREE	FROM	FUNCTION	FUSION
GENERAL	GET	GLOBAL	GOTO
GROUP	GROUPING	HANDLER	HASH
HOUR	IDENTITY	IF	IGNORE
IMMEDIATE	IN	INDICATOR	INITIALIZE
INITIALLY	INNER	INOUT	INPUT
INSERT	INT	INTEGER	INTERSECT
INTERSECTION	INTERVAL	INTO	IS
ISOLATION	ITERATE	JOIN	KEY
LANGUAGE	LARGE	LAST	LATERAL
LEADING	LEAVE	LEFT	LESS
LEVEL	LIKE	LIMIT	LOCAL
LOCALTIME	LOCALTIMESTAMP	LOCATOR	LOOP
MATCH	MEMBER	MEETS	MERGE
MINUTE	MODIFIES	MODIFY	MODULE
MONTH	MULTISET	NAMES	NATIONAL
NATURAL	NCHAR	NCLOB	NEW
NEXT	NO	NONE	NORMALIZE
NOT	NULL	NUMERIC	OBJECT
OF	OFF	OLD	ON
ONLY	OPEN	OPERATION	OPTION
OR	ORDER	ORDINALITY	OUT
OUTER	OUTPUT	PAD	PARAMETER
PARAMETERS	PARTIAL	PATH	PERIOD
POSTFIX	PRECEDES	PRECISION	PREFIX
PREORDER	PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVILEGES	PROCEDURE	PUBLIC
READ	READS	REAL	RECURSIVE
REDO	REF	REFERENCES	REFERENCING
RELATIVE	REPEAT	RESIGNAL	RESTRICT
RESULT	RETURN	RETURNS	REVOKE

RIGHT	ROLE	ROLLBACK	ROLLUP
ROUTINE	ROW	ROWS	SAVEPOINT
SCHEMA	SCROLL	SEARCH	SECOND
SECTION	SELECT	SEQUENCE	SESSION
SESSION_USER	SET	SETS	SIGNAL
SIZE	SMALLINT	SPECIFIC	SPECIFICTYPE
SQL	SQLEXCEPTION	SQLSTATE	SQLWARNING
START	STATE	STATIC	STRUCTURE
SUBMULTISET	SUCCEEDS	SUM	SYSTEM_USER
TABLE	TABLESAMPLE	TEMPORARY	TERMINATE
THAN	THEN	TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE	TO	TRAILING
TRANSACTION	TRANSLATION	TREAT	TRIGGER
TRUE	UESCAPE	UNDER	UNDO
UNION	UNIQUE	UNKNOWN	UNTIL
UPDATE	USAGE	USER	USING
VALUE	VALUES	VARCHAR	VARIABLE
VARYING	VIEW	WHEN	WHENEVER
WHERE	WHILE	WITH	WRITE
YEAR	ZONE		

Table B-3. DB2 keywords

ACQUIRE	ADD	AFTER	ALIAS
ALL	ALLOCATE	ALLOW	ALTER
AND	ANY	AS	ASC
ASUTIME	AUDIT	AUTHORIZATION	AUXILIARY
AVG	BEFORE	BEGIN	BETWEEN
BINARY	BUFFERPOOL	BY	CALL
CALLED	CAPTURE	CASCADED	CASE
CAST	CCSID	CHAR	CHARACTER
CHECK	CLOSE	CLUSTER	COLLECTION
COLLID	COLUMN	COMMENT	COMMIT
CONCAT	CONDITION	CONNECT	CONNECTION
CONSTRAINT	CONTAINS	CONTINUE	COUNT
COUNT_BIG	CREATE	CROSS	CURRENT
CURRENT_DATE	CURRENT_LC_PATH	CURRENT_PATH	CURRENT_SERVER
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_TIMEZONE	CURRENT_USER
CURSOR	DATA	DATABASE	DATE
DAY	DAYS	DB2GENERAL	DB2SQL
DBA	DBINFO	DBSPACE	DECLARE
DEFAULT	DELETE	DESC	DESCRIPTOR
DETERMINISTIC	DISALLOW	DISCONNECT	DISTINCT
DO	DOUBLE	DROP	DSSIZE

DYNAMIC	EDITPROC	ELSE	ELSEIF
END	END-EXEC	ERASE	ESCAPE
EXCEPT	EXCEPTION	EXCLUSIVE	EXECUTE
EXISTS	EXIT	EXPLAIN	EXTERNAL
FENCED	FETCH	FIELDPROC	FILE
FINAL	FOR	FOREIGN	FREE
FROM	FULL	FUNCTION	GENERAL
GENERATED	GO	GOTO	GRANT
GRAPHIC	GROUP	HANDLER	HAVING
HOUR	HOURS	IDENTIFIED	IF
IMMEDIATE	IN	INDEX	INDICATOR
INNER	INOUT	INSENSITIVE	INSERT
INTEGRITY	INTERSECT	INTO	IS
ISOBID	ISOLATION	JAVA	JOIN
KEY	LABEL	LANGUAGE	LC_CTYPE
LEAVE	LEFT	LIKE	LINKTYPE
LOCAL	LOCALE	LOCATOR	LOCATORS
LOCK	LOCKSIZE	LONG	LOOP
MAX	MICROSECOND	MICROSECONDS	MIN
MINUTE	MINUTES	MODE	MODIFIES
MONTH	MONTHS	NAME	NAMED
NHEADER	NO	NODENAME	NODENUMBER
NOT	NULL	NULLS	NUMPARTS
OBID	OF	ON	ONLY
OPEN	OPTIMIZATION	OPTIMIZE	OPTION
OR	ORDER	OUT	OUTER
PACKAGE	PAGE	PAGES	PARAMETER
PART	PARTITION	PATH	PCTFREE
PCTINDEX	PIECESIZE	PLAN	POSITION
PRECISION	PREPARE	PRIMARY	PRIQTY
PRIVATE	PRIVILEGES	PROCEDURE	PROGRAM
PSID	PUBLIC	QUERYNO	READ
READS	RECOVERY	REFERENCES	RELEASE
RENAME	REPEAT	RESET	RESOURCE
RESTRICT	RESULT	RETURN	RETURNS
REVOKE	RIGHT	ROLLBACK	ROW
ROWS	RRN	RUN	SCHEDULE
SCHEMA	SCRATCHPAD	SECOND	SECONDS
SECQTY	SECURITY	SELECT	SET
SHARE	SIMPLE	SOME	SOURCE
SPECIFIC	SQL	STANDARD	STATIC
STATISTICS	STAY	STOGROUP	STORES
STORPOOL	STYLE	SUBPAGES	SUBSTRING
SUM	SYNONYM	TABLE	TABLESPACE
THEN	TO	TRANSACTION	TRIGGER

TRIM	TYPE	UNDO	UNION
UNIQUE	UNTIL	UPDATE	USAGE
USER	USING	VALIDPROC	VALUES
VARIABLE	VARIANT	VCAT	VIEW
VOLUMES	WHEN	WHERE	WHILE
WITH	WLM	WORK	WRITE
YEAR	YEARS		

Table B-4. MySQL keywords

ACTION	ADD	AFTER	AGGREGATE
ALL	ALTER	AND	AS
ASC	AUTO_INCREMENT	AVG	AVG_ROW_LENGTH
BETWEEN	BIGINT	BINARY	BIT
BLOB	BOOL	BOTH	BY
CASCADE	CASE	CHANGE	CHAR
CHARACTER	CHECK	CHECKSUM	COLUMN
COLUMNS	COMMENT	CONSTRAINT	CREATE
CROSS	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
DATA	DATABASE	DATABASES	DATE
DATETIME	DAY	DAY_HOUR	DAY_MINUTE
DAY_SECOND	DAYOFMONTH	DAYOFWEEK	DAYOFYEAR
DEC	DECIMAL	DEFAULT	DELAY_KEY_WRITE
DELAYED	DELETE	DESC	DESCRIBE
DISTINCT	DISTINCTROW	DOUBLE	DROP
ELSE	ENCLOSED	END	ENUM
ESCAPE	ESCAPED	EXISTS	EXPLAIN
FIELDS	FILE	FIRST	FLOAT
FLOAT4	FLOAT8	FLUSH	FOR
FOREIGN	FROM	FULL	FUNCTION
GLOBAL	GRANT	GRANTS	GROUP
HAVING	HEAP	HIGH_PRIORITY	HOSTS
HOURL	HOURL_MINUTE	HOURL_SECOND	IDENTIFIED
IF	IGNORE	IN	INDEX
INFILE	INNER	INSERT	INSERT_ID
INT	INT1	INT2	INT3
INT4	INT8	INTEGER	INTERVAL
INTO	IS	ISAM	JOIN
KEY	KEYS	KILL	LAST_INSERT_ID
LEADING	LEFT	LENGTH	LIKE
LIMIT	LINES	LOAD	LOCAL
LOCK	LOGS	LONG	LOB
LONGTEXT	LOW_PRIORITY	MATCH	MAX
MAX_ROWS	MEDIUMLOB	MEDIUMINT	MEDIUMTEXT

MIDDLEINT	MIN_ROWS	MINUTE	MINUTE_SECOND
MODIFY	MONTH	MONTHNAME	MYISAM
NATURAL	NO	NOT	NULL
NUMERIC	ON	OPTIMIZE	OPTION
OPTIONALLY	OR	ORDER	OUTER
OUTFILE	PACK_KEYS	PARTIAL	PASSWORD
PRECISION	PRIMARY	PRIVILEGES	PROCEDURE
PROCESS	PROCESLIST	READ	REAL
REFERENCES	REGEXP	RELOAD	RENAME
REPLACE	RESTRICT	RETURNS	REVOKE
RLIKE	ROW	ROWS	SECOND
SELECT	SET	SHOW	SHUTDOWN
SMALLINT	SONAME	SQL_BIG_RESULT	SQL_BIG_SELECTS
SQL_BIG_TABLES	SQL_LOG_OFF	SQL_LOG_UPDATE	SQL_LOW_PRIORITY_UPDATES
SQL_SELECT_LIMIT	SQL_SMALL_RESULT	SQL_WARNINGS	STARTING
STATUS	STRAIGHT_JOIN	STRING	TABLE
TABLES	TEMPORARY	TERMINATED	TEXT
THEN	TIME	TIMESTAMP	TINYBLOB
TINYINT	TINYTEXT	TO	TRAILING
TYPE	UNIQUE	UNLOCK	UNSIGNED
UPDATE	USAGE	USE	USING
VALUES	VARBINARY	VARCHAR	VARIABLES
VARYING	WHEN	WITH	WRITE
ZEROFILL			

Table B-5. Oracle keywords

ACCESS	ADD	ALL	ALTER
AND	ANY	AS	ASC
AUDIT	BETWEEN	BY	CHAR
CHECK	CLUSTER	COLUMN	COMMENT
COMPRESS	CONNECT	CREATE	CURRENT
DATE	DECIMAL	DEFAULT	DELETE
DESC	DISTINCT	DROP	ELSE
EXCLUSIVE	EXISTS	FILE	FLOAT
FOR	FROM	GRANT	GROUP
HAVING	IDENTIFIED	IMMEDIATE	IN
INCREMENT	INDEX	INITIAL	INSERT
INTEGER	INTERSECT	INTO	IS
LEVEL	LIKE	LOCK	LONG
MAXEXTENTS	MINUS	MODE	MODIFY
MSLABEL	NOAUDIT	NOCOMPRESS	NOT
NOWAIT	NULL	NUMBER	OF
OFFLINE	ON	ONLINE	OPTION

OR	ORDER	PCTFREE	PRIOR
PRIVILEGES	PUBLIC	RAW	RENAME
RESOURCE	REVOKE	ROW	ROWID
ROWNUM	ROWS	SELECT	SESSION
SET	SHARE	SIZE	SMALLINT
START	SUCCESSFUL	SYNONYM	SYSDATE
TABLE	THEN	TO	TRIGGER
UID	UNION	UNIQUE	UPDATE
USER	VALIDATE	VALUES	VARCHAR
VARCHAR2	VIEW	WHENEVER	WHERE
WITH			

Table B-6. PostgreSQL keywords

ABORT	ADD	ALL	ALLOCATE
ALTER	ANALYZE	AND	ANY
ARE	AS	ASC	ASSERTION
AT	AUTHORIZATION	AVG	BEGIN
BETWEEN	BINARY	BIT	BIT_LENGTH
BOTH	BY	CASCADE	CASCADEDE
CASE	CAST	CATALOG	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CLOSE	CLUSTER	COALESCE	COLLATE
COLLATION	COLUMN	COMMIT	CONNECT
CONNECTION	CONSTRAINT	CONTINUE	CONVERT
COPY	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_SESSION
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATE	DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DELETE	DESC
DESCRIBE	DESCRIPTOR	DIAGNOSTICS	DISCONNECT
DISTINCT	DO	DOMAIN	DROP
ELSE	END	ESCAPE	EXCEPT
EXCEPTION	EXEC	EXECUTE	EXISTS
EXISTS	EXPLAIN	EXTEND	EXTERNAL
EXTRACT	FALSE	FETCH	FIRST
FLOAT	FOR	FOREIGN	FOUND
FROM	FULL	FULL	GET
GLOBAL	GO	GOTO	GRANT
GROUP	HAVING	IDENTITY	IN
INDICATOR	INNER	INPUT	INSERT
INTERSECT	INTERVAL	INTO	IS
JOIN	LAST	LEADING	LEFT
LIKE	LISTEN	LOAD	LOCAL

LOCK	LOWER	MAX	MIN
MODULE	MOVE	NAMES	NATIONAL
NATURAL	NCHAR	NEW	NO
NONE	NOT	NOTIFY	NULL
NULLIF	NUMERIC	OCTET_LENGTH	OFFSET
ON	OPEN	OR	ORDER
OUTER	OUTPUT	OVERLAPS	PARTIAL
POSITION	PRECISION	PREPARE	PRESERVE
PRIMARY	PRIVILEGES	PROCEDURE	PUBLIC
REFERENCES	RESET	REVOKE	RIGHT
ROLLBACK	ROWS	SCHEMA	SECTION
SELECT	SESSION	SESSION_USER	SET
SETOF	SHOW	SIZE	SOME
SQL	SQLCODE	SQLERROR	SQLSTATE
SUBSTRING	SUM	SYSTEM_USER	TABLE
TEMPORARY	THEN	TO	TO
TRAILING	TRANSACTION	TRANSLATE	TRANSLATION
TRIM	TRUE	UNION	UNIQUE
UNIQUE	UNKNOWN	UNLISTEN	UNTIL
UPDATE	UPPER	USAGE	USER
USING	VACUUM	VALUE	VALUES
VARCHAR	VARYING	VERBOSE	VIEW
WHEN	WHENEVER	WHERE	WITH
WORK	WRITE		

Table B-7. Microsoft SQL Server keywords

ADD	ALL	ALTER	AND
ANY	AS	ASC	AUTHORIZATION
BACKUP	BEGIN	BETWEEN	BREAK
BROWSE	BULK	BY	CASCADE
CASE	CHECK	CHECKPOINT	CLOSE
CLUSTERED	COALESCE	COLLATE	COLUMN
COMMIT	COMPUTE	CONSTRAINT	CONTAINS
CONTAINSTABLE	CONTINUE	CONVERT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DBCC	DEALLOCATE	DECLARE	DEFAULT
DELETE	DENY	DESC	DISK
DISTINCT	DISTRIBUTED	DOUBLE	DROP
DUMMY	DUMP	ELSE	END
ERRLVL	EXCEPT	EXEC	EXECUTE
EXISTS	EXIT	FETCH	FILE
FILLFACTOR	FOR	FOREIGN	FREETEXT

FREETEXTTABLE	FROM	FULL	FUNCTION
GOTO	GRANT	GROUP	HAVING
HOLDLOCK	IDENTITY	IDENTITY_INSERT	IDENTITYCOL
IF	IN	INDEX	INNER
INSERT	INTERSECT	INTO	IS
JOIN	KEY	KILL	LEFT
LIKE	LINENO	LOAD	NATIONAL
NOCHECK	NONCLUSTERED	NOT	NULL
NULLIF	OF	OFF	OFFSETS
ON	OPEN	OPENDATASOURCE	OPENQUERY
OPENROWSET	OPENXML	OPTION	OR
ORDER	OUTER	PERCENT	PLAN
PRECISION	PRIMARY	PRINT	PROC
PROCEDURE	PUBLIC	RAISERROR	READ
READTEXT	RECONFIGURE	REFERENCES	REPLICATION
RESTORE	RESTRICT	RETURN	REVOKE
RIGHT	ROLLBACK	ROWCOUNT	ROWGUIDCOL
RULE	SAVE	SCHEMA	SELECT
SESSION_USER	SET	SETUSER	SHUTDOWN
SOME	STATISTICS	SYSTEM_USER	TABLE
TEXTSIZE	THEN	TO	TOP
TRAN	TRANSACTION	TRIGGER	TRUNCATE
TSEQUAL	UNION	UNIQUE	UPDATE
UPDATETEXT	USE	USER	VALUES
VARYING	VIEW	WAITFOR	WHEN
WHERE	WHILE	WITH	WRITETEXT

Table B-8. Sybase Adaptive Server keywords

ABSOLUTE	ACTION	ADD	ALL
ALLOCATE	ALTER	AND	ANY
ARE	ARITH_OVERFLOW	AS	ASC
ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH
BOTH	BREAK	BROWSE	BULK
BY	CASCADE	CASCADED	CASE
CAST	CATALOG	CHAR	CHAR_CONVERT
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CHECKPOINT	CLOSE	CLUSTERED	COALESCE
COLLATE	COLLATION	COLUMN	COMMIT
COMPUTE	CONFIRM	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONTINUE	CONTROLROW
CONVERT	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME

CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DATE	DAY	DBCC	DEALLOCATE
DEC	DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE	DESC
DESCRIBE	DESCRIPTOR	DETERMINISTIC	DIAGNOSTICS
DISCONNECT	DISK DISTINCT	DOMAIN	DOUBLE
DROP	DUMMY	DUMP	ELSE
END	END-EXEC	ENDTRAN	ERRLVL
ERRORDATA	ERROREXIT	ESCAPE	EXCEPT
EXCEPTION	EXCLUSIVE	EXEC	EXECUTE
EXISTS	EXIT	EXP_ROW_SIZE	EXTERNAL
EXTRACT	FALSE	FETCH	FILLFACTOR
FIRST	FLOAT	FOR	FOREIGN
FOUND	FROM	FULL	FUNC
FUNCTION	GET	GLOBAL	GO
GOTO	GRANT	GROUP	HAVING
HOLDLOCK	HOUR	IDENTITY	IDENTITY_GAP
IDENTITY_INSERT	IDENTITY_START	IF	IMMEDIATE
IN	INDEX	INDICATOR	INITIALLY
INNER	INOUT	INPUT	INSENSITIVE
INSERT	INSTALL	INT	INTEGER
INTERSECT	INTERVAL	INTO	IS
ISOLATION	JAR	JOIN	KEY
KILL	LANGUAGE	LAST	LEADING
LEFT	LEVEL	LIKE	LINENO
LOAD	LOCAL	LOCK	LOWER
MATCH	MAX	MAX_ROWS_PER_PAGE	MIN
MINUTE	MIRROR	MIRROREXIT	MODIFY
MODULE	MONTH	NAMES	NATIONAL
NATURAL	NCHAR	NEW	NEXT
NO	NOHOLDLOCK	NONCLUSTERED	NOT
NULL	NULLIF	NUMERIC	NUMERIC_TRUNCATION
OCTET_LENGTH	OF	OFF	OFFSETS
ON	ONCE	ONLINE	ONLY
OPEN	OPTION	OR	ORDER
OUT	OUTER	OUTPUT	OVER
OVERLAPS	PAD	PARTIAL	PARTITION
PERM	PERMANENT	PLAN	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY
PRINT	PRIOR	PRIVILEGES	PROC
PROCEDURE	PROCESSEXIT	PROXY_TABLE	PUBLIC
QUIESCE	RAISERROR	READ	READPAST
READTEXT	REAL	RECONFIGURE	REFERENCES REMOVE
RELATIVE	REORG	REPLACE	REPLICATION
RESERVEPAGEGAP	RESTRICT	RETURN	RETURNS

REVOKE	RIGHT	ROLE	ROLLBACK
ROWCOUNT	ROWS	RULE	SAVE
SCHEMA	SCROLL	SECOND	SECTION
SELECT	SESSION_USER	SET	SETUSER
SHARED	SHUTDOWN	SIZE	SMALLINT
SOME	SPACE	SQL	SQLCODE
SQLERROR	SQLSTATE	STATISTICS	STRINGSIZE
STRIPE	SUBSTRING	SUM	SYB_IDENTITY
SYB_RESTREE	SYB_TERMINATE	SYSTEM_USER	TABLE
TEMP	TEMPORARY	TEXTSIZE	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRAN	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER	TRIM
TRUE	TRUNCATE	TSEQUAL	UNION
UNIQUE	UNKNOWN	UNPARTITION	UPDATE
UPPER	USAGE	USE	USER
USER_OPTION	USING	VALUE	VALUES
VARCHAR	VARYING	VIEW	WAITFOR
WHEN	WHENEVER	WHERE	WHILE
WITH	WORK	WRITE	WRITETEXT
YEAR	ZONE		

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *SQL in a Nutshell* is a chameleon. There are approximately 85 species of chameleons existing in the world today. They are mostly indigenous to Africa, although there are a few species found in Asia and in Europe. Most are tree dwellers. The chameleon is relatively small; the average adult size is between 6 and 12 inches. It lives mostly on insects and uses its long tongue to capture its prey. Indeed, the tongue is a critical tool. It can stretch up to 1.5 times the lizard's body length, and there is an adhesive pad on the end on which the insects are trapped. There are several other characteristics common to all species of chameleons. For example, its eyes are large and protruding, and the lizard can see 360 degrees without moving its head or body. Its toes are on either side of its feet, usually three on one side and two on the other. This is ideal for moving quickly and efficiently through tree branches.

Chameleons are best known for their ability to change their appearance to adapt to their physical environment. Actually, several types of reptiles can change their skin color, but the chameleon is far and away the most accomplished. This skill, which is moderated by the nervous system, obviously is invaluable for hunting prey and avoiding predators, and also helps to stabilize body temperature. The extent of this camouflage capability is related to the gender, age, and species of the lizard.

Mary Brady was the production editor for *SQL in a Nutshell*, and Jane Ellin was the copyeditor. Ann Schirmer was the proofreader. Sarah Sherman and Colleen Gorman provided quality control. Julie Hawks wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Julie Hawks to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Mary Brady.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]



< Day Day Up >



[**SYMBOL**] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

/ division arithmetic operator
!> not greater than operator
!< not less than operator
!≡ not equal to operator
(pound symbol)
(double-pound symbol)
% modula arithmetic operator
& bitwise AND operator
> greater than operator
>= greater than or equal to operator
< less than operator
<= less than or equal to operator
* multiplication arithmetic operator
+ addition arithmetic operator
+ unary operator
- subtraction arithmetic operator
- unary operator
≡ assignment operator
≡ equal to comparison operator
^ bitwise exclusive OR operator
| bitwise OR operator
~ unary operator





[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[ABS function](#)

[ABSTIME function \(PostgreSQL\)](#)

[ABSVAL function \(DB2\)](#)

[ACOS function](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

[ADD_MONTHS function \(Oracle\)](#)

[ADDDATE function \(MySQL\)](#)

[addition \(+\) arithmetic operator](#)

[ADO.NET](#)

[_beginning transactions](#)

[_bound parameters](#)

[_closing connection](#)

[_committing transactions](#)

[_database connection](#)

[_database program example](#)

[_DataReader methods](#)

[_Get](#)

[_error handling](#)

[_executing statements](#)

[_parameter object types](#)

[_retrieving data](#)

[_rolling back transactions](#)

[AGE function \(PostgreSQL\)](#)

[aggregate functions](#)

[ALL operator 2nd](#)

[_keywords](#)

[_platform differences](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[ALTER DATABASE statement](#) [See CREATE/ALTER DATABASE statements]

[ALTER DOMAIN statement, quick reference](#)

[ALTER FUNCTION statement](#) [See CREATE/ALTER FUNCTION/PROCEDURE statements]

[ALTER INDEX statement](#) [See CREATE/ALTER INDEX statements]

[ALTER METHOD statement](#) [See CREATE/ALTER METHOD statements]

[ALTER PROCEDURE statement](#) [See CREATE/ALTER FUNCTION/PROCEDURE statements]

[ALTER TABLE statement](#) [See CREATE/ALTER TABLE statements]

[ALTER TRIGGER statement](#) [See CREATE/ALTER TRIGGER statements]

[ALTER TYPE statement](#) [See CREATE/ALTER TYPE statements]

[ALTER VIEW statement](#) [See CREATE/ALTER VIEW statements]

[AND operator](#)

[ANSI standard](#)

[AnsiString parameter object type \(ADO.NET\)](#)

[AnsiStringFixedLength parameter object type \(ADO.NET\)](#)

[ANY operator 2nd](#)

[_keywords](#)

[_platform differences](#)

[_platform support for](#)

- [_programming_tips](#)
- [_quick_reference](#)
- [_rules](#)
- [_SQL2003_syntax](#)
- [APP_NAME function](#)
- [_SQL_Server](#)
- [AREA function \(PostgreSQL\)](#)
- [arithmetic operators 2nd](#)
- [AS keyword and assignment operator](#)
- [ASCII function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- [ASCIISTR function \(Oracle\)](#)
- [ASIN function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- [assignment \(=\) operator](#)
- [assignment operators](#)
- [ATAN function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- [ATAN2 function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [ATANH function \(DB2\)](#)
- [ATN2 function \(SQL Server\)](#)
- [AuthorizationID](#)
- [AVG function 2nd](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[BENCHMARK function \(MySQL\)](#)

[BETWEEN clause](#)

[BETWEEN operator 2nd](#)

[_keywords](#)

[_platform differences](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[BFILE datatype](#)

[_Oracle](#)

[_platform comparison](#)

[BFILENAME function \(Oracle\)](#)

[BIGINT datatype](#)

[_DB2](#)

[_MySQL](#)

[_platform comparison](#)

[_SQL Server 2nd](#)

[BIGINT function \(DB2\)](#)

[BIGSERIAL datatype](#)

[_platform comparison](#)

[_PostgreSQL 2nd](#)

[BIN function \(MySQL\)](#)

[BIN_TO_NUM function \(Oracle\)](#)

[BINARY datatype](#)

[_platform comparison](#)

[_SQL Server 2nd](#)

[_Sybase adaptive server](#)

[BINARY function \(MySQL\)](#)

[Binary parameter object type \(ADO.NET\)](#)

[BINARY_CHECKSUM function](#)

[_SQL Server](#)

[BINARY_DOUBLE datatype](#)

[_Oracle](#)

[_platform comparison](#)

[BINARY_FLOAT datatype](#)

[_Oracle](#)

[_platform comparison](#)

[BIT datatype](#)

[_MySQL](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[BIT VARYING datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[BIT_AND function \(MySQL\)](#)

[BIT_COUNT function \(MySQL\)](#)

[BIT_LENGTH function](#)

[BIT_OR function \(MySQL\)](#)

[BITAND function \(Oracle\)](#)

[bitwise AND \(&\) operator](#)

[bitwise exclusive OR \(^\) operator](#)

[bitwise operators](#)

[bitwise OR \(|\) operator](#)

BLOB datatype

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[BLOB function \(DB2\)](#)

BOOL datatype

[_MySQL](#)

[_platform comparison](#)

[_PostgreSQL](#)

[Boolean comparison operators](#)

BOOLEAN datatype

[_platform comparison](#)

[_PostgreSQL](#)

[Boolean parameter object type \(ADO.NET\)](#)

bound parameters

[_ADO.NET](#)

[_JDBC](#)

BOX datatype

[_platform comparison](#)

[_PostgreSQL](#)

[BOX function \(PostgreSQL\)](#)

[BROADCAST function \(PostgreSQL\)](#)

[built-in scalar functions](#)

[BULKADMIN \(SQL Server system role\)](#)

[Byte parameter object type \(ADO.NET\)](#)

BYTEA datatype

[_platform comparison](#)

[_PostgreSQL](#)





[SYMBOL] [A] [B] [**C**] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[CALL statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)
- [_versus EXECUTE statement](#)

[candidate key](#)

[CARDINALITY function \(Oracle\)](#)

[CASE function](#)

[CAST function 2nd](#)

[catalogs](#)

[CBRT function \(PostgreSQL\)](#)

[CEIL function](#)

[CENTER function \(PostgreSQL\)](#)

[Chaos isolation level \(Transaction object \(ADO.NET\)\)](#)

[CHAR datatype](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[CHAR FOR BIT DATA datatype](#)

- [_DB2](#)
- [_platform comparison](#)

[CHAR function](#)

- [_DB2](#)
- [_MySQL](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[CHAR VARYING datatype](#)

- [_DB2](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_SQL Server](#)

[CHAR\(\) datatype](#)

- [_SQL Server](#)
- [_Sybase adaptive server](#)

[CHAR_LENGTH function 2nd](#)

- [_PostgreSQL](#)

[CHARACTER datatype](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[character sets](#)

[character string literals](#)

CHARACTER VARYING datatype

[_DB2](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[CHARINDEX function \(SQL Server\)](#)

[CHARTOROWID function \(Oracle\)](#)

[CHECK constraints](#)

[CHECKSUM function](#)

[_SQL Server](#)

[CHECKSUM_AGG function](#)

[_SQL Server](#)

CHR function

[_DB2](#)

[_Oracle](#)

CIDR datatype

[_platform comparison](#)

[_PostgreSQL](#)

CIRCLE datatype

[_platform comparison](#)

[_PostgreSQL](#)

[CIRCLE function \(PostgreSQL\)](#)

CLOB datatype

[_DB2](#)

[_Oracle](#)

[_platform comparison](#)

[CLOB function \(DB2\)](#)

[CLOSE CURSOR statement](#)

[_DB2](#)

[_keywords](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

clusters

[COALESCE function](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

Codd, E. F.

[COL_LENGTH function \(SQL Server\)](#)

[COL_NAME function \(SQL Server\)](#)

collations

[COLLECT function \(Oracle\)](#)

column-level constraints

[column_constraint_clause \(Oracle\)](#)

columns

[COMMIT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

- [_platform_support_for](#)
- [_PostgreSQL](#)
- [_programming_tips](#)
- [_quick_reference](#)
- [_rules](#)
- [_SQL_Server](#)
- [_SQL2003_syntax](#)
- [COMMIT TO SWITCHOVER TO](#)
- [COMPARE function](#)
- [comparison_operators_2nd](#)
- [COMPOSE function \(Oracle\)](#)
- [COMPRESS function \(MySQL\)](#)
- [compressed_tables_in_Oracle](#)
- [CONCAT function](#)
 - [_DB2](#)
 - [_Oracle](#)
- [CONCAT_WS function \(MySQL\)](#)
- [concatenation_operator](#)
- [CONNECT statement](#)
 - [_DB2](#)
 - [_keywords](#)
 - [_Oracle](#)
 - [_platform_support_for](#)
 - [_PostgreSQL](#)
 - [_programming_tips](#)
 - [_quick_reference](#)
 - [_rules](#)
 - [_SQL_Server](#)
 - [_SQL2003_syntax](#)
- [CONNECTION_ID\(.\) function \(MySQL\)](#)
- [connections \(database\)](#)
 - [_closing](#)
 - [_opening](#)
 - [_ADO.NET](#)
 - [_JDBC](#)
- [connectivity_establishing](#)
- [constraints](#)
 - [_CHECK](#)
 - [_FOREIGN_KEY](#)
 - [_PRIMARY_KEY](#)
 - [_scope](#)
 - [_syntax](#)
 - [_UNIQUE](#)
- [CONTAINS function](#)
 - [_SQL_Server](#)
- [CONTAINSTABLE function](#)
 - [_SQL_Server](#)
- [CONV function \(MySQL\)](#)
- [CONVERT function](#)
 - [_Oracle](#)
 - [_SQL_Server](#)
- [CORR function_2nd](#)
- [CORR_K function \(Oracle\)](#)
- [correlated_subqueries](#)
- [Correlated_subquery_in_WHERE/HAVING_clause](#)
- [COS function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)

- [_PostgreSQL](#)
- [_SQL_Server](#)
- COSH function
 - [_DB2](#)
 - [_Oracle](#)
- COT function
 - [_DB2](#)
 - [_MySQL](#)
 - [_PostgreSQL_2nd](#)
 - [_SQL_Server](#)
- COUNT function_2nd
- COUNT_BIG function
- COVAR_POP function_2nd
- COVAR_SAMP function_2nd
- CREATE/ALTER DATABASE statements
 - [_DB2](#) [See DB2, CREATE/ALTER DATABASE statements]
 - [_MySQL](#)
 - [_Oracle](#) [See Oracle, CREATE/ALTER DATABASE statements]
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference_2nd](#)
 - [_rules](#)
 - [_SQL_Server](#) [See SQL Server, CREATE/ALTER DATABASE statements]
- CREATE/ALTER DOMAIN statements, [quick reference](#)
- CREATE/ALTER FUNCTION/PROCEDURE statements
 - [_DB2](#) [See DB2, CREATE/ALTER FUNCTION/PROCEDURE statements]
 - [_keywords](#)
 - [_CALL ON NULL INPUT](#)
 - [_CASCADE](#)
 - [_code_block](#)
 - [_CONTAINS SQL](#)
 - [_CREATE](#)
 - [_DETERMINISTIC](#)
 - [_DYNAMIC RESULT SETS](#)
 - [_IN](#)
 - [_LANGUAGE](#)
 - [_MODIFIES SQL DATA](#)
 - [_NAME](#)
 - [_NO SQL](#)
 - [_OUT](#)
 - [_PARAMETER STYLE](#)
 - [_READS SQL DATA](#)
 - [_RESTRICT](#)
 - [_RETURN NULL ON NULL INPUT](#)
 - [_RETURNS](#)
 - [_SPECIFIC](#)
 - [_STATIC DISPATCH](#)
 - [_MySQL](#)
 - [_Oracle](#) [See Oracle, CREATE/ALTER FUNCTION/PROCEDURE statements]
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference_2nd](#)
 - [_rules](#)
 - [_SQL_Server](#) [See SQL Server, CREATE/ALTER FUNCTION/PROCEDURE statements]
 - [_SQL2003 syntax](#)
- CREATE/ALTER INDEX statements
 - [_common vendor syntax](#)

- [_DB2](#) [See DB2, CREATE/ALTER INDEX statements]
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#) [See Oracle, CREATE/ALTER INDEX statements]
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference 2nd](#)
- [_rules](#)
- [_SQL Server](#) [See SQL Server, CREATE/ALTER INDEX statements]

[CREATE/ALTER METHOD statements](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference 2nd](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[CREATE/ALTER ROLE statements](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[CREATE/ALTER SCHEMA statements](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[CREATE/ALTER TABLE statements](#)

- [_DB2](#) [See DB2, CREATE/ALTER TABLE statements]
- [_keywords](#)
 - [_ADD 2nd](#)
 - [_ALTER](#)
 - [_CASCADE](#)
 - [_column_name](#)
 - [_CONSTRAINT](#)
 - [_DROP COLUMN](#)
 - [_DROP CONSTRAINT](#)
 - [_GLOBAL TEMPORARY](#)
 - [_LIKE](#)
 - [_LOCAL TEMPORARY](#)

[_OF](#)

[_ON COMMIT](#)

[_REF IS](#)

[_RESTRICT](#)

[_MySQL](#) [See MySQL, CREATE/ALTER TABLE statements]

[_Oracle](#) [See Oracle, CREATE/ALTER TABLE statements]

[_platform support for](#)

[_PostgreSQL](#) [See PostgreSQL, CREATE/ALTER TABLE statements]

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#) [See SQL Server, CREATE/ALTER TABLE statements]

[_SQL2003 syntax](#)

[CREATE/ALTER TRIGGER statements](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#) [See Oracle, CREATE/ALTER TRIGGER statements]

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference 2nd](#)

[_rules](#)

[_SQL Server \(SQL Server, CREATE/ALTER TRIGGER statements\)](#)

[_SQL2003 syntax](#)

[CREATE/ALTER TYPE statements](#)

[_DB2](#) [See DB2, CREATE/ALTER TYPE statements]

[_keywords](#)

[_MySQL](#)

[_Oracle](#) [See Oracle, CREATE/ALTER TYPE statements]

[_platform support for](#)

[_PostgreSQL](#) [See PostgreSQL, CREATE/ALTER TYPE statements]

[_programming tips](#)

[_quick reference 2nd](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[CREATE/ALTER VIEW statements](#)

[_DB2](#) [See DB2, CREATE/ALTER VIEW statements]

[_keywords](#)

[_MySQL](#)

[_Oracle](#) [See Oracle, CREATE/ALTER VIEW statements]

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference 2nd](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[CROSS JOIN](#)

[CUME_DIST function 2nd 3rd](#)

[CURDATE\(.\) function \(MySQL\)](#)

[Currency parameter object type \(ADO.NET\)](#)

[CURSOR datatype](#)

[_platform comparison](#)

[_SQL Server](#)

[cursors](#) [See CLOSE CURSOR statement DECLARE CURSOR command]

[CURTIME\(.\) function \(MySQL\)](#)

[CV function \(Oracle\)](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

data

[_retrieving](#)

[_ADO.NET](#)

[_JDBC](#)

[Data Control Language \(DCL\)](#)

[Data Definition Language \(DDL\)](#)

[Data Manipulation Language \(DML\)](#)

[data structures](#)

[database objects and operators 2nd](#)

[database programming](#) [See programming databases]

[DATABASE\(\) function \(MySQL\)](#)

[DATABASEPROPERTYEX function](#)

[_SQL Server](#)

[DATALENGTH function \(SQL Server\)](#)

[DATALINK datatype](#)

[_DB2](#)

[_platform comparison](#)

[DataReader methods \(ADO.NET\)](#)

[_Get](#)

[datatypes](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

[_Sybase adaptive server](#)

[DATE datatype](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[DATE function \(DB2\)](#)

[DATE_FORMAT function \(MySQL\)](#)

[DATE_ADD function \(MySQL\)](#)

[DATE_PART function \(PostgreSQL\)](#)

[DATE_SUB function \(MySQL\)](#)

[DATE_TRUNC function \(PostgreSQL\)](#)

[DATEADD function \(SQL Server\)](#)

[DATEDIFF function \(SQL Server\)](#)

[DATENAME function \(SQL Server\)](#)

[DATEPART function \(SQL Server\)](#)

[DATETIME datatype](#)

[_MySQL](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server 2nd](#)

[_Sybase adaptive server](#)

[DateTime parameter object type \(ADO.NET\)](#)

[DAY function](#)

[_DB2](#)

[_SQL Server](#)

[DAYNAME function](#)

[_DB2](#)

MySQL

DAYOFMONTH function (MySQL)

DAYOFWEEK function

DB2

MySQL

DAYOFWEEK_ISO function (DB2)

DAYOFYEAR function

DB2

MySQL

DAYS function (DB2)

DB_ACCESSADMIN (SQL Server system role)

DB_BACKUPOPERATOR (SQL Server system role)

DB_DATAREADER (SQL Server system role)

DB_DATAWRITER (SQL Server system role)

DB_DDLADMIN (SQL Server system role)

DB_DENYDATAREADER (SQL Server system role)

DB_DENYDATAWRITER (SQL Server system role)

DB_ID function (SQL Server)

DB_NAME function (SQL Server)

DB_OWNER (SQL Server system role)

DB_SECURITYADMIN (SQL Server system role)

DB2

_allowed_characters_in_identifiers

_beginning_identifier_character

_CALL statement

_CLOSE CURSOR statement

_COMMIT statement

_CONNECT statement

_connecting_using JDBC

_CREATE/ALTER DATABASE statements

_ALIAS

_APPLY

_AT DBPARTITIONNUM

_AUTOCONFIGURE

_CATALOG TABLESPACE

_COLLATE USING

_CREATE {DATABASE | DB}

_DFT_EXTENT_SZ

_EXTENTSIZE

_MANAGED BY

_NUMSEGS

_ON

_OVERHEAD

_PREFETCHSIZE

_tablespace_definition

_TEMPORARY TABLESPACE

_TRANSFERRATE

_USER TABLESPACE

_USING CODESET

_WITH

_CREATE/ALTER FUNCTION/PROCEDURE statements

_CARDINALITY

_CONTAINS SQL

_DBINFO

_EXTERNAL NAME

_FENCED

_FINAL CALL

_INHERIT SPECIAL REGISTERS

_LANGUAGE

MODIFIES SQL DATA
NO SQL
PARALLEL
PARAMETER STYLE
PREDICATES
PROGRAM TYPE
READS SQL DATA
RETURNS
SCRATCHPAD
THREADSAFE
TRANSFORM GROUP
CREATE/ALTER INDEX statements
ALLOW REVERSE SCANS
ASC
CLUSTER
COLLECT STATISTICS
DESC
EXTEND USING
INCLUDE
MINPCTUSED
PCTFREE
SPECIFICATION ONLY
CREATE/ALTER METHOD statements
CREATE/ALTER ROLE statements
CREATE/ALTER SCHEMA statements
CREATE/ALTER TABLE statements
ACTIVATE NOT LOGGED INITIALLY
ACTIVATE VALUE COMPRESSION
ADD
ALTER [COLUMN]
ALTER {CHECK | FOREIGN KEY}
APPEND
AS
AS IDENTITY
COMPACT
COMPRESS SYSTEM DEFAULT
copy_options
DATA CAPTURE 2nd
DEFINITION ONLY
derived_table_options
DROP
FILE LINK CONTROL
GENERATED
IN
intermediate table, defining
LIKE
LINKTYPE URL
LOCKSIZE
LOGGED
MODE DB2OPTIONS
NLINE LENGTH
NO LINK CONTROL
NOT LOGGED INITIALLY
OPTIONS
ORGANIZE BY
PARTITIONING KEY
PCTFREE
REPLICATED
SCOPE

- [_SET MATERIALIZED_QUERY](#)
- [_UNDER](#)
- [_VALUE COMPRESSION](#)
- [_VOLATILE](#)
- [_WITH RESTRICT ON DROP](#)
- [CREATE/ALTER TRIGGER statements](#)
- [CREATE/ALTER TYPE statements](#)
 - [_ADD METHOD](#)
 - [_ALTER METHOD](#)
 - [_DISTINCT](#)
 - [_DROP METHOD](#)
 - [_FENCED](#)
 - [_INLINE LENGTH](#)
 - [_METHOD](#)
 - [_MODE DB2SQL](#)
 - [_OVERRIDING](#)
 - [_RESTRICT](#)
 - [_THREADSAFE](#)
 - [_WITH COMPARISONS](#)
 - [_WITHOUT COMPARISONS](#)
- [CREATE/ALTER VIEW statements](#)
 - [_WITH CHECK OPTION clause](#)
- [datalink-related attributes](#)
- [datatypes](#)
- [DECLARE CURSOR command](#)
- [DELETE statement](#)
- [DISCONNECT statement](#)
- [DROP statements](#)
 - [_\[DISTINCT\] TYPE](#)
 - [_\[SPECIFIC\] FUNCTION](#)
 - [_\[SPECIFIC\] METHOD](#)
 - [_BUFFERPOOL](#)
 - [_DATABASE PARTITION GROUP](#)
 - [_EVENT MONITOR](#)
 - [_FUNCTION MAPPING](#)
 - [_INDEX](#)
 - [_NICKNAME](#)
 - [_PACKAGE](#)
 - [_RESTRICT](#)
 - [_SCHEMA](#)
 - [_SEQUENCE](#)
 - [_SERVER](#)
 - [_TABLE](#)
 - [_TABLESPACE](#)
 - [_TRANSFORM](#)
 - [_TRIGGER](#)
 - [_TYPE MAPPING](#)
 - [_USER MAPPING FOR](#)
 - [_VIEW](#)
 - [_WRAPPER](#)
- [EXCEPT set operator](#)
- [external scalar user-defined functions](#)
- [external stored procedure](#)
- [external table user-defined functions](#)
- [FETCH statement](#)
- [functions](#)
 - [_ABSVAL](#)
 - [_ACOS](#)
 - [_ASCII](#)

ASIN
ATAN
ATAN2
ATANH
BIGINT
BLOB
CHAR
CHR
CLOB
COALESCE
CONCAT
COS
COSH
COT
DATE
DAY
DAYNAME
DAYOFWEEK
DAYOFWEEK_ISO
DAYOFYEAR
DAYS
DBCLOB
DBPARTITIONNUM
DEC
DECIMAL
DECRYPT_BIN
DECRYPT_CHAR
DEGREES
DEREF
DIFFERENCE
DIGITS
DOUBLE
ENCRYPT
EVENT_MON_STATE
FLOAT
GENERATE_UNIQUE()
GETHINT
GRAPHIC
HASHEDVALUE
HEX
HOUR
IDENTITY_VAL_LOCAL()
INSERT
INTEGER
JULIAN_DAY
LEFT
LENGTH
LOCATE
LOG
LOG10
LONG_VARCHAR
LONG_VARGRAPHIC
LTRIM
MICROSECOND
MIDNIGHT_SECONDS
MINUTE
MONTH
MONTHNAME
MULTIPLY_ALT

NULLIF
POSSTR
QUARTER
RADIANS
RAISE_ERROR
RAND
REAL
REC2XML
REPEAT
REPLACE
RIGHT
ROUND
RTRIM
SECOND
SIGN
SIN
SINH
SMALLINT
SOUNDEX
SPACE
STDDEV
SUBSTR
TABLE_NAME
TABLE_SCHEMA
TAN
TANH
TIME
TIMESTAMP
TIMESTAMP_FORMAT
TIMESTAMP_ISO
TIMESTAMPDIFF
TO_CHAR
TO_DATE
TRANSLATE
TRUNC
TRUNCATE
TYPE_ID
TYPE_NAME
TYPE_SCHEMA
VALUE
VAR
VARCHAR
VARCHAR_FORMAT
VARGRAPHIC
VARIANCE
WEEK
WEEK_ISO
YEAR

GRANT statement
{ALTER | USAGE} ON SEQUENCE
{ALTERIN | CREATEIN | DROPIN} ON SCHEMA
{BIND | CONTROL | EXECUTE} ON PACKAGE
CONTROL ON INDEX
database_privileges
EXECUTE ON
object_privilege
PASSTHRU ON SERVER
TO USER
USE OF TABLESPACE

- [_WITH GRANT OPTION](#)
- [_identifier size](#)
- [INSERT statement](#)
- [INTERSECT set operator](#)
- [JOIN subclause](#)
- [keywords](#)
- [LDAP](#)
- [LIKE operator](#)
- [MERGE statement](#)
- [OLE-DB external table user-defined functions](#)
- [OPEN statement](#)
- [ORDER BY clause](#)
- [RELEASE SAVEPOINT statement](#)
- [reserved identifiers](#)
- [RETURN statement](#)
- [REVOKE statement](#)
 - [_ { ALTERIN | CREATEIN | DROPIN } ON SCHEMA](#)
 - [_ { BIND | CONTROL | EXECUTE } ON PACKAGE](#)
 - [_ { ALTER | USAGE } ON SEQUENCE](#)
 - [_CONTROL ON INDEX](#)
 - [_database_privileges](#)
 - [_EXECUTE ON](#)
 - [_FROM](#)
 - [_levels of system privileges](#)
 - [_object_privilege](#)
 - [_PASSTHRU ON SERVER](#)
 - [_RESTRICT](#)
 - [_USE OF TABLESPACE](#)
- [ROLLBACK statement](#)
- [SAVEPOINT statement](#)
- [schema addressing](#)
- [SELECT statement](#)
- [SET CONNECTION statement](#)
- [SET CONSTRAINT statement](#)
- [SET PATH statement](#)
- [SET ROLE statement](#)
- [SET SCHEMA statement](#)
- [SET SESSION AUTHORIZATION statement](#)
- [SET statement](#)
- [SET TIME ZONE statement](#)
- [SET TRANSACTION statement](#)
- [sourced user-defined functions](#)
- [SQL stored procedure](#)
- [SQL user-defined functions](#)
- [START TRANSACTION statement](#)
- [SUBQUERY substatement](#)
- [TRUNCATE TABLE statement](#)
- [UNION set operator](#)
- [UPDATE statement](#)
- [web site](#)
- [WHERE clause](#)
- [window syntax](#)

DB2 keywords

DBCLOB datatype

- [_DB2](#)
- [_platform comparison](#)

DBCLOB function (DB2)

DBCREATOR (SQL Server system role)

DBPARTITIONNUM function (DB2)

[DBTIMEZONE function \(Oracle\)](#)

DEC datatype

[_DB2](#)

[_platform_comparison](#)

[_SQL_Server](#)

[DEC function \(DB2\)](#)

DECIMAL datatype

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform_comparison](#)

[_PostgreSQL_2nd](#)

[_SQL_Server](#)

[DECIMAL function \(DB2\)](#)

[Decimal parameter object type \(ADO.NET\)](#)

[declarative processing](#)

[DECLARE CURSOR command](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

DECODE function

[_MySQL](#)

[_Oracle](#)

[DECOMPOSE function \(Oracle\)](#)

[DECRYPT_BIN function \(DB2\)](#)

[DECRYPT_CHAR function \(DB2\)](#)

DEGREES function

[_DB2](#)

[_MySQL](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[DELETE statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_operators_and_2nd](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

[delimited identifiers](#)

[delimiters](#)

[DENSE_RANK function 2nd_3rd](#)

[DEPTH function \(Oracle\)](#)

DEREF function

[_DB2](#)

[_Oracle](#)

- [deterministic functions](#)
- [dialects, SQL](#)
- [DIAMETER function \(PostgreSQL\)](#)
- [DIFFERENCE function](#)
 - [_DB2](#)
 - [_SQL Server](#)
- [DIGITS function \(DB2\)](#)
- [DISCONNECT statement](#)
 - [_DB2](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference](#)
 - [_rules](#)
 - [_SQL Server](#)
 - [_SQL2003 syntax](#)
- [DISKADMIN \(SQL Server system role\)](#)
- [division \(/\) arithmetic operator](#)
- [DOUBLE datatype](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_platform comparison](#)
- [DOUBLE function \(DB2\)](#)
- [Double parameter object type \(ADO.NET\)](#)
- [DOUBLE PRECISION datatype](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform comparison](#)
 - [_PostgreSQL](#)
 - [_SQL Server](#)
- [double-pound symbol \(##\)](#)
- [DROP DATABASE statement, quick reference](#)
- [DROP DOMAIN statement, quick reference](#)
- [DROP FUNCTION statement, quick reference](#)
- [DROP INDEX statement, quick reference](#)
- [DROP METHOD statement, quick reference](#)
- [DROP ROLE statement, quick reference](#)
- [DROP statements](#)
 - [_DB2](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference](#)
 - [_rules](#)
 - [_SQL Server](#)
 - [_SQL2003 syntax](#)
- [DROP TABLE statement, quick reference](#)
- [DROP TRIGGER statement, quick reference](#)
- [DROP TYPE statement, quick reference](#)
- [DROP VIEW statement, quick reference](#)
- [DUMP function \(Oracle\)](#)



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[ELT function \(MySQL\)](#)

[EMPTY_BLOB\(.\) function \(Oracle\)](#)

[EMPTY_CLOB\(.\) function \(Oracle\)](#)

[ENCODE function \(MySQL\)](#)

ENCRYPT function

[_DB2](#)

[_MySQL](#)

ENUM datatype

[_MySQL](#)

[_platform comparison](#)

[equal to \(=\) comparison operator](#)

error handling

[_ADO.NET](#)

[_JDBC](#)

[EVENT_MON_STATE function \(DB2\)](#)

EXCEPT set operator

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[EXECUTE statement versus CALL statement](#)

EXISTS operator 2nd

[_platform differences](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[EXISTSNODE function \(Oracle\)](#)

EXP function

explicit transactions

[EXPORT_SET function \(MySQL\)](#)

expressions

[_operator precedence and](#)

[_testing against range of values](#)

EXTRACT function

[_Oracle](#)

[EXTRACTVALUE function \(Oracle\)](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[FETCH statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[FIELD function \(MySQL\)](#)

[FILE_ID function](#)

- [_SQL Server](#)

[FILE_NAME function](#)

- [_SQL Server](#)

[FILEGROUP_ID function](#)

- [_SQL Server](#)

[FILEGROUP_NAME function](#)

- [_SQL Server](#)

[FILEGROUPPROPERTY function](#)

- [_SQL Server](#)

[FILEPROPERTY function](#)

- [_SQL Server](#)

[filtering checks](#)

[FIND_IN_SET function \(MySQL\)](#)

[FIRST function \(Oracle\)](#)

[FIRST_VALUE function \(Oracle\)](#)

[FLOAT datatype](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_SQL Server 2nd](#)

[FLOAT function](#)

- [_DB2](#)
- [_PostgreSQL](#)

[FLOAT4 datatype](#)

- [_platform comparison](#)
- [_PostgreSQL](#)

[FLOAT4 function \(PostgreSQL\)](#)

[FLOAT8 datatype](#)

- [_platform comparison](#)
- [_PostgreSQL](#)

[FLOOR function](#)

[FOREIGN KEY constraints](#)

- [_CASCADE keyword](#)
- [_FOREIGN KEY keyword](#)
- [_MATCH keyword](#)
- [_NO ACTION keyword](#)
- [_ON DELETE keyword](#)
- [_ON UPDATE keyword](#)
- [_REFERENCES keyword](#)

- [_RESTRICT keyword](#)
- [_SET_DEFAULT keyword](#)
- [_SET_NULL keyword](#)
- [FORMAT function \(MySQL\)](#)
- [FORMATMESSAGE function](#)
 - [_SQL Server](#)
- [freeing_resources](#)
- [FREETEXT function \(SQL Server\)](#)
- [FREETEXTTABLE function](#)
 - [_SQL Server](#)
- [FROM clause](#)
- [FROM_DAYS function \(MySQL\)](#)
- [FROM_TZ function \(Oracle\)](#)
- [FROM_UNIXTIME function \(MySQL\) 2nd](#)
- [FULL \[OUTER\] JOIN](#)
- [FULLTEXTCATALOGPROPERTY function](#)
 - [_SQL Server](#)
- [FULLTEXTSERVICEPROPERTY function](#)
 - [_SQL Server](#)
- [functions](#)
 - [_ABS](#)
 - [_aggregate](#)
 - [_APP_NAME\(.\)](#)
 - [_AVG 2nd](#)
 - [_BINARY_CHECKSUM](#)
 - [_BIT_LENGTH](#)
 - [_CASE](#)
 - [_CAST 2nd](#)
 - [_CEIL](#)
 - [_CHAR_LENGTH 2nd](#)
 - [_CHECKSUM](#)
 - [_CHECKSUM_AGG](#)
 - [_COALESCE](#)
 - [_COMPARE](#)
 - [_CONTAINS](#)
 - [_CONTAINSTABLE](#)
 - [_CONVERT](#)
 - [_CORR 2nd](#)
 - [_COUNT 2nd](#)
 - [_COUNT_BIG](#)
 - [_COVAR_POP 2nd](#)
 - [_COVAR_SAMP 2nd](#)
 - [_CUME_DIST 2nd](#)
 - [_CUME_DIST\(.\)](#)
 - [_DATABASEPROPERTYEX](#)
 - [_DAY](#)
 - [_DB2 supported \[See DB2, functions\]](#)
 - [_DENSE_RANK 2nd](#)
 - [_DENSE_RANK\(.\)](#)
 - [_deterministic](#)
 - [_EXP](#)
 - [_EXTRACT](#)
 - [_FILE_ID](#)
 - [_FILE_NAME](#)
 - [_FILEGROUP_ID](#)
 - [_FILEGROUP_NAME](#)
 - [_FILEGROUPPROPERTY](#)
 - [_FILEPROPERTY](#)
 - [_FLOOR](#)

[_FORMATMESSAGE](#)
[_FREETEXTTABLE](#)
[_FULLTEXTCATALOGPROPERTY](#)
[_FULLTEXTSERVICEPROPERTY](#)
[_GETANSINULL](#)
[_GETUTCDATE\(\)](#)
[_GROUPING](#)
[_HEXTOINT](#)
[_IDENT_CURRENT](#)
[_IDENT_INCR](#)
[_IDENT_SEED](#)
[_IDENTITY](#)
[_INDEXPROPERTY](#)
[_INTTOHEX](#)
[_IS_MEMBER](#)
[_IS_SEC_SERVICE_ON](#)
[_IS_SRVROLEMEMBER](#)
[_ISDATE](#)
[_ISNUMERIC](#)
[_lct_admin](#)
[_LEFT](#)
[_LEN](#)
[_LICENSE_ENABLED](#)
[_LN](#)
[_LOWER](#)
[_MAX 2nd](#)
[_MIN 2nd](#)
[_MOD](#)
[_MONTH](#)
[_MUT_EXCL_ROLES](#)
[_MySQL supported \[See MySQL, functions\]](#)
[_NCHAR](#)
[_NEWID\(\)](#)
[_nondeterministic](#)
[_NULLIF](#)
[_numeric](#)
[_OBJECTPROPERTY](#)
[_OCTET_LENGTH](#)
[_OPEN](#)
[_OPENDATASOURCE](#)
[_OPENQUERY](#)
[_OPENROWSET](#)
[_operators and 2nd](#)
[_Oracle supported \[See Oracle, functions\]](#)
[_OVERLAY](#)
[_PARSENAME](#)
[_PATINDEX](#)
[_PERCENT_RANK 2nd 3rd](#)
[_PERCENTILE_CONT 2nd](#)
[_PERCENTILE_DISC 2nd](#)
[_PERMISSIONS](#)
[_POSITION](#)
[_PostgreSQL \[See PostgreSQL, functions\]](#)
[_POWER](#)
[_PROC_ROLE](#)
[_RANK 2nd](#)
[_RANK\(\)](#)
[_REGR_AVGX 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)
[_REGR_AVGY 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[_REGR_COUNT 2nd 3rd 4th 5th 6th 7th 8th](#)
[_REGR_INTERCEPT 2nd 3rd 4th 5th 6th 7th](#)
[_REGR_R2 2nd 3rd 4th 5th 6th](#)
[_REGR_SLOPE 2nd 3rd 4th 5th](#)
[_REGR_SXX 2nd 3rd 4th](#)
[_REGR_SXY 2nd 3rd](#)
[_REGR_SYY 2nd](#)
[_REPLACE](#)
[_ROLE_CONTAIN](#)
[_ROLE_ID](#)
[_ROLE_NAME](#)
[_ROW_NUMBER](#)
[_ROWCNT](#)
[_ROWCOUNT_BIG\(.\)](#)
[_scalar 2nd](#)
[_built-in](#)
[_categories of](#)
[_SESSION_USER](#)
[_SHOW_ROLE\(.\)](#)
[_SHOW_SEC_SERVICES\(.\)](#)
[_SORTKEY](#)
[_SQL Server supported \[See SQL Server, functions\]](#)
[_SQL2003_numeric](#)
[_SQRT](#)
[_STATS_DATE](#)
[_STDDEV_POP 2nd](#)
[_STDDEV_SAMP 2nd](#)
[_STDEV](#)
[_STDEVP](#)
[_string](#)
[_SUBSTRING](#)
[_SUM 2nd](#)
[_SUSER_ID](#)
[_SUSER_NAME](#)
[_SUSER_SID](#)
[_SUSER_SNAME](#)
[_SYB_SENDSMSG](#)
[_Sybase adaptive server versus SQL Server](#)
[_Sybase adaptive server-supported](#)
[_SYSTEM_USER](#)
[_TO_UNICHAR](#)
[_TRANSLATE](#)
[_TRIM](#)
[_TSEQUEL](#)
[_TYPEPROPERTY](#)
[_types of](#)
[_UHIGHSURR](#)
[_UOWSURR](#)
[_UNICODE](#)
[_UPPER](#)
[_USCALAR](#)
[_VALID_NAME](#)
[_VALID_USER](#)
[_VAR](#)
[_VAR_POP 2nd](#)
[_VAR_SAMP 2nd](#)
[_VARP](#)
[_WIDTH_BUCKET](#)
[_window 2nd](#)

windowing

YEAR



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[GENERATE_UNIQUE\(\) function \(DB2\)](#)

[GET DIAGNOSTIC statement](#)

[Get methods \(ADO.NET DataReader\)](#)

[Get methods \(JDBC ResultSet\)](#)

[GET_LOCK function \(MySQL\)](#)

[GETANSINULL function](#)

[_SQL Server](#)

[GETDATE\(\) function \(SQL Server\)](#)

[GETHINT function \(DB2\)](#)

[GETUTCDATE\(\) function](#)

[_SQL Server](#)

[GRANT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[GRAPHIC datatype](#)

[_DB2](#)

[_platform comparison](#)

[GRAPHIC function \(DB2\)](#)

[greater than \(>\) operator](#)

[greater than or equal to \(>=\) operator](#)

[GREATEST function](#)

[_MySQL](#)

[_Oracle](#)

[GROUP BY clause 2nd](#)

[group clause](#)

[GROUP_ID\(\) function \(Oracle\)](#)

[GROUPING function](#)

[_Oracle](#)

[_SQL Server](#)

[GROUPING_ID function \(Oracle\)](#)



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[HASHEDVALUE function \(DB2\)](#)

[HAVING clause](#)

[HEIGHT function \(PostgreSQL\)](#)

HEX function

[_DB2](#)

[_MySQL](#)

[HEXTOINT function](#)

[HEXTORAW function \(Oracle\)](#)

[HOST function \(PostgreSQL\)](#)

[HOST_ID\(.\) function \(SQL Server\)](#)

[HOST_NAME\(.\) function \(SQL Server\)](#)

HOUR function

[_DB2](#)

[_MySQL](#)



< Day Day Up >



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[IDENT_CURRENT function](#)

[_SQL Server](#)

[IDENT_INCR function](#)

[_SQL Server](#)

[IDENT_SEED function](#)

[_SQL Server](#)

[identifier rules](#)

[identifiers](#)

[_allowed_characters](#)

[_beginning_characters](#)

[_quoted](#)

[_reserved](#)

[_size](#)

[IDENTITY function](#)

[_SQL Server](#)

[IDENTITY_VAL_LOCAL\(\) function \(DB2\)](#)

[IF function \(MySQL\)](#)

[IFNULL function \(MySQL\)](#)

[IMAGE datatype](#)

[_platform_comparison](#)

[_SQL Server](#)

[implicit transactions](#)

[IN operator 2nd](#)

[_keywords](#)

[_platform_support_for](#)

[_quick_reference](#)

[_SQL2003_syntax](#)

[INDEX_COL function \(SQL Server\)](#)

[INDEXPROPERTY function](#)

[_SQL Server](#)

[INET datatype](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[INIT.ORA file](#)

[INITCAP function](#)

[_Oracle](#)

[INITCAP function \(PostgreSQL\)](#)

[INNER JOIN](#)

[INSERT function](#)

[_DB2](#)

[_MySQL](#)

[INSERT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_operators_and_2nd](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003_syntax](#)

[INSTR function](#)

[_MySQL](#)

[_Oracle](#)

[INT datatype](#)

[_DB2](#)

[_MySQL](#)

[_platform_comparison](#)

[_SQL_Server](#)

[Int parameter object type \(ADO.NET\)](#)

[INT2 datatype](#)

[_platform_comparison](#)

[INT4 datatype](#)

[_platform_comparison](#)

[INT8 datatype](#)

[_PostgreSQL](#)

[INTEGER datatype](#)

[_DB2](#)

[_Oracle](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[INTEGER function](#)

[_DB2](#)

[_PostgreSQL](#)

[INTERSECT set operator](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

[INTERVAL datatype](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[INTERVAL DAY datatype](#)

[_Oracle](#)

[INTERVAL DAY TO SECOND datatype](#)

[_platform_comparison](#)

[INTERVAL function](#)

[_MySQL](#)

[_PostgreSQL](#)

[INTERVAL YEAR datatype](#)

[_Oracle](#)

[INTERVAL YEAR TO MONTH datatype](#)

[_platform_comparison](#)

[INTTOHEX function](#)

[IS operator](#)

[_keywords](#)

[_platform_support_for](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL2003_syntax](#)

[IS_FREE_LOCK function \(MySQL\)](#)

[IS_MEMBER function](#)

[_SQL_Server](#)
[IS_SEC_SERVICE_ON function](#)
[IS_SRVROLEMEMBER function](#)
[_SQL_Server](#)
[ISCLOSED function \(PostgreSQL\)](#)
[ISDATE function](#)
[_SQL_Server](#)
[ISFINITE function \(PostgreSQL\)](#)
[ISNULL function](#)
[_MySQL](#)
[_SQL_Server](#)
[ISNUMERIC function](#)
[_SQL_Server](#)
[ISOPEN function \(PostgreSQL\)](#)
[ITERATION_NUMBER function \(Oracle\)](#)





[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[Java Virtual Machine \(JVM\) class loader](#)

JDBC

- [_beginning_transactions](#)
- [_bound_parameters](#)
- [_closing_connection](#)
- [_committing_transactions](#)
- [_database_connection](#)
- [_database_program_example](#)
- [_error_handling](#)
- [_executing_statements](#)
- [_PreparedStatement_set_methods](#)
- [_ResultSet_methods](#)
- [_Get](#)
- [_retrieving_data](#)
- [_rolling_back_transactions](#)

[JDBC \(Java Database Connectivity\)](#)

[JOIN clause 2nd](#)

[JOIN subclause 2nd](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform_support_for](#)
- [_PostgreSQL](#)
- [_programming_tips](#)
- [_rules](#)
- [_SQL_Server](#)
- [_SQL2003_syntax](#)
- [_types](#)

[JULIAN_DAY function \(DB2\)](#)





< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

keywords 2nd

.DB2 2nd

.Microsoft SQL Server

.MySQL 2nd

.Oracle 2nd

.PostgreSQL 2nd

.shared 2nd

.SQL Server

.SQL2003 2nd

.Sybase adaptive server 2nd 3rd 4th 5th



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[LAG function \(Oracle\)](#)

[LAST function \(Oracle\)](#)

[LAST_DAY function \(Oracle\)](#)

[LAST_INSERT_ID function \(MySQL\)](#)

[LAST_VALUE function \(Oracle\)](#)

[LCASE function \(MySQL\)](#)

[lct_admin function](#)

[LDAP \(Lightweight Directory Access Protocol\)](#)

[_on DB2 servers](#)

[LEAD function \(Oracle\)](#)

[LEAST function](#)

[_MySQL](#)

[_Oracle](#)

[LEFT \[OUTER\] JOIN](#)

[LEFT function](#)

[_DB2](#)

[_MySQL](#)

[_SQL Server](#)

[LEN function](#)

[_SQL Server](#)

[LENGTH function](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[LENGTHB function \(Oracle\)](#)

[less than \(<\) operator](#)

[less than or equal to \(<=\) operator](#)

[LICENSE_ENABLED function](#)

[LIKE operator 2nd](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[LINE datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[literals 2nd](#)

[LN function](#)

[LNNVL function \(Oracle\)](#)

[LOAD_FILE function \(MySQL\)](#)

[LOCALTIMESTAMP function \(Oracle\)](#)

[LOCATE function](#)

[_DB2](#)

[_MySQL](#)

[LOCATE\(substr,str,pos\)](#)

[LOG function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL_Server](#)

LOG10 function

- [_DB2](#)
- [_MySQL](#)
- [_SQL_Server](#)

[LOG2 function \(MySQL\)](#)

[logfiles \(Oracle\)](#)

[logical operators 2nd](#)

LONG datatype

- [_Oracle](#)
- [_platform comparison](#)

LONG RAW datatype

- [_Oracle](#)
- [_platform comparison](#)

LONG VARCHAR datatype

- [_DB2](#)
- [_platform comparison](#)

LONG VARCHAR FOR BIT DATA datatype

- [_DB2](#)

LONG VARGRAPHIC datatype

- [_DB2](#)
- [_platform comparison](#)

[LONG_VARCHAR function \(DB2\)](#)

[LONG_VARGRAPHIC function \(DB2\)](#)

LONGBLOB datatype

- [_MySQL](#)
- [_platform comparison](#)

LONGTEXT datatype

- [_MySQL](#)
- [_platform comparison](#)

[LOWER function](#)

LPAD function

- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)

LSEG datatype

- [_platform comparison](#)
- [_PostgreSQL](#)

[LSEG function \(PostgreSQL\)](#)

LTRIM function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL_Server](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

MACADDR datatype

[_platform_comparison](#)

[_PostgreSQL](#)

[MAKE_REF function \(Oracle\)](#)

[MAKE_SET function \(MySQL\)](#)

[MASKLEN function \(PostgreSQL\)](#)

[MAX function 2nd](#)

[MD5 function \(MySQL\)](#)

[MEDIAN function \(Oracle\)](#)

MEDIUMBLOB datatype

[_MySQL](#)

[_platform_comparison](#)

MEDIUMINT datatype

[_MySQL](#)

MEDIUMTEXT datatype

[_MySQL](#)

[_platform_comparison](#)

[MERGE statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

[metadata](#)

[methods](#)

[_ADO.NET_DataReader](#)

[_JDBC_ResultSet](#)

[MICROSECOND function \(DB2\)](#)

[Microsoft SQL Server](#)

[Microsoft SQL Server keywords](#)

[MID function \(MySQL\)](#)

[MIDNIGHT_SECONDS function \(DB2\)](#)

[MIN function 2nd](#)

MINUTE function

[_DB2](#)

[_MySQL](#)

[MOD function](#)

[modula \(%\) arithmetic operator \(SQLServer\)](#)

MONEY datatype

[_platform_comparison](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[MONTH function](#)

[_DB2](#)

[_MySQL](#)

[_SQL_Server](#)

MONTHNAME function

[_DB2](#)

MySQL

MONTHS_BETWEEN function (Oracle)

multiplication (*) arithmetic operator

MULTIPLY_ALT function (DB2)

MUT_EXCL_ROLES function

MySQL

_allowed characters in identifiers

_ALTER TABLE syntax

_beginning identifier character

_CALL statement

_COMMIT statement

_connecting using JDBC

_CREATE/ALTER DATABASE statements

_CREATE/ALTER FUNCTION/PROCEDURE statements

_CREATE/ALTER INDEX statements

_CREATE/ALTER METHOD statements

_CREATE/ALTER ROLE statements

_CREATE/ALTER SCHEMA statements

CREATE/ALTER TABLE

_keywords and parameters

_CREATE/ALTER TABLE statements

_ALTER

_ALTER COLUMN

_AUTO_INCREMENT

_AVG_ROW_LENGTH

_CHANGE

_CHECKSUM

_COMMENT

_constraint_type

_DATA DIRECTORY

_DELAY_KEY_WRITE

_DROP

_ENABLE KEYS

_IF NOT EXISTS

_IGNORE SELECT_statement

_INDEX DIRECTORY

_INSERT_METHOD

_keywords

_MAX_ROWS

_MIN_ROWS

_MODIFY

_PACK_KEYS

_parameters

_PASSWORD

_RAID_CHUNKS

_RAID_CHUNKSIZE

_RAID_TYPE

_RENAME

_REPLACE SELECT_statement

_ROW_FORMAT

_TEMPORARY

_TYPE

_CREATE/ALTER TRIGGER statements

_CREATE/ALTER TYPE statements

_CREATE/ALTER VIEW statements

_datatypes

_DECLARE CURSOR command

_DELETE statement

_DISCONNECT statement

DROP statements
EXCEPT set operator
FETCH statement
functions
_,POW
_ACOS
_ADDDATE
_ASCII
_ASIN
_ATAN
_ATAN2
_BENCHMARK
_BIN
_BINARY
_BIT_AND
_BIT_COUNT
_BIT_OR
_CHAR
_COALESCE
_COMPRESS
_CONCAT_WS
_CONNECTION_ID(.)
_CONV
_COS
_COT
_CURDATE(.)
_CURTIME(.)
_DATABASE(.)
_DATE_FORMAT
_DATE_ADD
_DATE_SUB
_DAYNAME
_DAYOFMONTH
_DAYOFWEEK
_DAYOFYEAR
_DECODE
_DEGREES
_ELT
_ENCODE
_ENCRYPT
_EXPORT_SET
_FIELD
_FIND_IN_SET
_FORMAT
_FROM_DAYS
_FROM_UNIXTIME 2nd
_GET_LOCK
_GREATEST
_HEX
_HOUR
_IF
_IFNULL
_INSERT
_INSTR
_INTERVAL
_IS_FREE_LOCK
_ISNULL
_LAST_INSERT_ID
_LCASE

LEAST
LEFT
LENGTH
LOAD_FILE
LOCATE
LOG
LOG10
LOG2
LPAD
LTRIM
MAKE_SET
MD5
MID
MINUTE
MONTH
MONTHNAME
NOW()
NULLIF
OCT
ORD
PASSWORD
PERIOD_ADD
PERIOD_DIFF
PI()
POSITION
POWER
QUARTER
RADIANS
RAND()
RELEASE_LOCK
REPEAT
REPLACE
REVERSE
RIGHT
ROUND
RPAD
RTRIM
SEC_TO_TIME
SECOND
SESSION_USER()
SHA
SHA1
SIGN
SIN
SOUNDEX
SPACE
STD
STDDEV
STRCMP
SUBDATE
SUBSTRING
SUBSTRING_INDEX
SYSDATE()
SYSTEM_USER()
TAN
TIME_FORMAT
TIME_TO_SEC
TO_DAYS
TRUNCATE

- [_UCASE](#)
- [_UNCOMPRESS](#)
- [_UNHEX](#)
- [_UNIX_TIMESTAMP](#)
- [_USER\(\)](#)
- [_VERSION\(\)](#)
- [_WEEK](#)
- [_WEEKDAY](#)
- [_YEAR](#)
- [_YEARWEEK](#)
- [_GRANT statement](#)
- [_identifier size](#)
- [_INSERT statement](#)
- [_INTERSECT set operator](#)
- [_JOIN subclause](#)
- [_keywords](#)
- [_LIKE operator](#)
- [_MERGE statement](#)
- [_OPEN statement](#)
- [_ORDER BY clause](#)
- [_RELEASE SAVEPOINT statement](#)
- [_reserved identifiers](#)
- [_RETURN statement](#)
- [_REVOKE statement](#)
- [_ROLLBACK statement](#)
- [_SAVEPOINT statement](#)
- [_schema addressing](#)
- [_SELECT statement](#)
 - [_{FOR UPDATE | LOCK IN SHARE MODE}](#)
 - [_FROM É USE INDEX](#)
 - [_HIGH_PRIORITY](#)
 - [_IGNORE INDEX](#)
 - [_INTO](#)
 - [_JOIN syntax](#)
 - [_LIMIT](#)
 - [_PROCEDURE](#)
 - [_select_item](#)
 - [_SQL_BIG_RESULT](#)
 - [_SQL_BUFFER_RESULT](#)
 - [_SQL_SMALL_RESULT](#)
 - [_STRAIGHT_JOIN](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_special rules](#)
- [_START TRANSACTION statement](#)
- [_SUBQUERY substatement](#)
- [_TRUNCATE TABLE statement](#)
- [_UNION set operator](#)
- [_UPDATE statement](#)
- [_web site](#)
- [_WHERE clause](#)



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[naming conventions](#)

[NANVL function \(Oracle\)](#)

NATIONAL CHAR datatype

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL CHAR VARYING datatype

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL CHARACTER datatype

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL CHARACTER VARYING datatype

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL TEXT datatype

[_platform comparison](#)

[_SQL Server](#)

[NATURAL \[INNER | {LEFT | RIGHT}\[OUTER\] \] JOIN](#)

NCHAR datatype

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

[NCHAR function](#)

[_Oracle](#)

[_SQL Server](#)

NCHAR VARYING datatype

[_Oracle](#)

[_platform comparison](#)

NCHAR() datatype

[_SQL Server 2nd](#)

[_Sybase adaptive server](#)

NCLOB datatype

[_Oracle](#)

[_platform comparison](#)

[nested parentheses and operator precedence](#)

nested tables

[_in FROM clause](#)

[_Oracle](#)

[_subqueries](#)

[NETMASK function \(PostgreSQL\)](#)

[NETWORK function \(PostgreSQL\)](#)

[NEW_TIME function \(Oracle\)](#)

[NEWID\(\) function](#)

[_SQL Server](#)

[NEXT_DAY function \(Oracle\)](#)

[NLS_CHARSET_DECL_LEN function \(Oracle\)](#)

[NLS_CHARSET_ID function \(Oracle\)](#)

[NLS_CHARSET_NAME function \(Oracle\)](#)

- [NLS_INITCAP function \(Oracle\)](#)
- [NLS_LOWER function \(Oracle\)](#)
- [NLS_UPPER function \(Oracle\)](#)
- [NLSSORT function \(Oracle\)](#)
- [nondeterministic functions](#)
- [not equal to \(!=\) operator 2nd 3rd](#)
- [not greater than \(!>\) operator](#)
- [not less than \(!<\) operator](#)
- [NOT operator](#)
- [NOW\(\) function \(MySQL\)](#)
- [NPOINTS function \(PostgreSQL\)](#)
- [NTEXT datatype](#)
 - [_platform comparison](#)
 - [_SQL Server 2nd](#)
- [NTILE function \(Oracle\)](#)
- [NULL values](#)
- [NULLIF function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL Server](#)
- [NUM datatype](#)
 - [_DB2](#)
- [NUMBER datatype](#)
 - [_Oracle](#)
 - [_platform comparison](#)
- [NUMERIC datatype](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform comparison](#)
 - [_PostgreSQL 2nd](#)
 - [_SQL Server 2nd](#)
- [numeric datatype and unary operators](#)
- [numeric functions](#)
 - [_enhanced](#)
- [NUMTODSINTERVAL function \(Oracle\)](#)
- [NUMTOYMINTERVAL function \(Oracle\)](#)
- [NVARCHAR\(\) datatype \(Sybase Adaptive Server\)](#)
- [NVARCHAR datatype](#)
 - [_MySQL](#)
 - [_platform comparison](#)
 - [_SQL Server](#)
- [NVARCHAR2 datatype](#)
 - [_Oracle](#)
- [NVARCHAR2\(\) datatype](#)
 - [_platform comparison](#)
- [NVL function \(Oracle\)](#)
- [NVL2 function \(Oracle\)](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[OBJECT_ID function \(SQL Server\)](#)

[OBJECT_NAME function \(SQL Server\)](#)

[OBJECTPROPERTY function](#)

[_SQL Server](#)

[objects](#)

[OCT function \(MySQL\)](#)

[OCTET_LENGTH function](#)

[OdbcConnection](#)

[_connection_string_attributes_for](#)

[OID datatype](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[OLAP functions](#)

[OleDbConnection](#)

[_connection_string_attributes_for](#)

[OPEN function](#)

[_SQL Server](#)

[OPEN statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003_syntax](#)

[OPENDATASOURCE function](#)

[_SQL Server](#)

[OPENQUERY function](#)

[_SQL Server](#)

[OPENROWSET function](#)

[_SQL Server](#)

[operator precedence](#)

[operators 2nd](#)

[_string](#)

[OPERATORS, quick reference](#)

[OR operator](#)

[ORA_HASH function \(Oracle\)](#)

[Oracle](#)

[_allowed_characters_in_identifiers](#)

[_ALTER INDEX syntax](#)

[_ALTER TABLE statement](#)

[_ALTER TABLE syntax](#)

[_altering_a_partitioned_table](#)

[_basics](#)

[_beginning_identifier_character](#)

[_CALL statement](#)

[_column_constraint_clause](#)

[_COMMIT statement](#)

[_compressed_tables](#)

[_CONNECT statement](#)

[_connecting_using JDBC](#)
[_CREATE_INDEX syntax](#)
[_CREATE/ALTER DATABASE statements](#)
[_ACTIVATE](#)
[_ADD LOGFILE](#)
[_ADD SUPPLEMENTAL LOG DATA](#)
[_ALLOW](#)
[_ARCHIVELOG 2nd](#)
[_AUTOMATIC](#)
[_BACKUP CONTROLFILE TO](#)
[_BEGIN BACKUP](#)
[_BIGFILE](#)
[_CANCEL](#)
[_CHARACTER SET](#)
[_CLEAR LOGFILE](#)
[_CONTINUE](#)
[_CONTROLFILE REUSE](#)
[_CREATE DATAFILE](#)
[_CREATE STANDBY CONTROLFILE AS](#)
[_DATABASE](#)
[_DATAFILE 2nd](#)
[_DEFAULT TABLESPACE](#)
[_DEFAULT TEMPORARY TABLESPACE](#)
[_ENABLE BLOCK CHANGE TRACKING](#)
[_ENABLE THREAD](#)
[_EXTENT MANAGEMENT 2nd](#)
[_FLASHBACK](#)
[_FOR](#)
[_FORCE LOGGING 2nd](#)
[_GUARD](#)
[_LOGFILE 2nd](#)
[_MANAGED STANDBY DATABASE](#)
[_MAXDATAFILES](#)
[_MAXINSTANCES](#)
[_MAXLOGFILES](#)
[_MAXLOGHISTORY](#)
[_MAXLOGMEMBERS](#)
[_MOUNT](#)
[_NATIONAL CHARACTER SET](#)
[_OPEN](#)
[_PARALLEL](#)
[_RECOVER](#)
[_REGISTER](#)
[_RENAME FILE](#)
[_RENAME GLOBAL_NAME TO](#)
[_SET DEFAULT TABLESPACE](#)
[_SET STANDBY](#)
[_SET TIME_ZONE](#)
[_SMALLFILE](#)
[_STANDBY](#)
[_START LOGICAL STANDBY APPLY](#)
[_SYSAUX DATAFILE](#)
[_TABLESPACE](#)
[_TEST](#)
[_UNDO TABLESPACE](#)
[_UNTIL](#)
[_USER SYS IDENTIFIED BY](#)
[_USER SYSTEM IDENTIFIED BY](#)
[_CREATE/ALTER FUNCTION/PROCEDURE statements](#)

AGGREGATE
ALTER
AS
AUTHID
code_block
COMPILE
CREATE
IN
IS
LANGUAGE
NOCOPY
OUT
PARALLEL_ENABLE
PIPELINED
CREATE/ALTER INDEX statements
ALLOCATE EXTENT
alter_index_partitioning_clause
ASC
BITMAP
CLUSTER
COALESCE
COMPRESS
COMPUTE
DEALLOCATE UNUSED
DESC
ENABLE
GLOBAL
INDEXTYPE IS
LOCAL
LOGGING
MONITORING USAGE
NOSORT
PARALLEL
PARAMETERS
physical_attributes_clause
rebuild_clause
RENAME TO
REVERSE
SHRINK SPACE
TABLESPACE DEFAULT
UNUSABLE
UPDATE BLOCK REFERENCES
CREATE/ALTER METHOD statements
CREATE/ALTER ROLE statements
CREATE/ALTER SCHEMA statements
CREATE/ALTER TABLE statements
ADD É
ADD PARTITION
ALLOCATE EXTENT
AS object_type
AS subquery
CACHE
CLUSTER
COALESCE
COALESCE PARTITION
COMPRESS
constraint_state_clause
DEALLOCATE UNUSED
DROP É

DROP PARTITION
ENABLE ROW MOVEMENT
ENABLE TABLE LOCK
EXCHANGE PARTITION
foreign_key_constraint
GROUP
inline_ref_constraint 2nd
LOGGING
MERGE PARTITION
MINIMIZE RECORDS_PER_BLOCK
MODIFY DEFAULT ATTRIBUTES
MODIFY É
MODIFY PARTITION
MODIFY SUBPARTITION
MONITORING
MOVE É
MOVE PARTITION
OBJECT IDENTIFIER IS
OF XMLTYPE
OIDINDEX
ON COMMIT
ORGANIZATION EXTERNAL
ORGANIZATION HEAP
ORGANIZATION INDEX
PARALLEL
parameters
partition_clause
partition_level_subpartition_clause
partition_storage_clause
physical_attributes_clause
PROJECT COLUMN
RENAME É
RENAME PARTITION
ROWDEPENDENCIES
SET SUBPARTITION TEMPLATE
SET UNUSED É
SHRINK SPACE
SPLIT PARTITION
storage_clause 2nd
SUBSTITUTABLE AT ALL LEVELS
table_constraint_clause
table_partition_description_clause
table_ref_constraint 2nd
TABLESPACE
TRUNCATE PARTITION
update_index_clause
UPGRADE INCLUDING DATA
xml_schema_spec
XMLTYPE
CREATE/ALTER TRIGGER statements
CREATE/ALTER TYPE statements
ADD
AS OBJECT
AS TABLE OF
AS VARRAY
AUTHID
BODY
CASCADE
COMPILE

[_compiler_directives_2nd](#)
[_CONVERT_TO_SUBSTITUTABLE](#)
[_datatype](#)
[_DEBUG](#)
[_DROP](#)
[_EXCEPTIONS INTO](#)
[_EXTERNAL NAME](#)
[_FORCE](#)
[_INCLUDING TABLE DATA](#)
[_INVALIDATE](#)
[_MEMBER](#)
[_MODIFY](#)
[_MODIFY_ELEMENT_TYPE](#)
[_MODIFY LIMIT](#)
[_OID](#)
[_OR REPLACE](#)
[_OVERRIDING](#)
[_REPLACE AS OBJECT](#)
[_REUSE SETTINGS](#)
[_SPECIFICATION](#)
[_STATIC](#)
[_UNDER](#)
[_CREATE/ALTER VIEW statements](#)
[_datatypes](#)
[_DECLARE CURSOR command](#)
[_default database user accounts](#)
[_DELETE statement](#)
[_DISCONNECT statement](#)
[_DROP statements](#)
[_\[PUBLIC\] DATABASE LINK](#)
[_\[PUBLIC\] SYNONYM](#)
[_CLUSTER](#)
[_CONTEXT](#)
[_DATABASE](#)
[_DIMENSION](#)
[_DIRECTORY](#)
[_FUNCTION](#)
[_INDEX](#)
[_INDEXTYPE](#)
[_JAVA](#)
[_LIBRARY](#)
[_MATERIALIZED VIEW](#)
[_OPERATOR](#)
[_OUTLINE](#)
[_PACKAGE](#)
[_PROCEDURE](#)
[_PROFILE](#)
[_ROLE](#)
[_ROLLBACK SEGMENT](#)
[_SEQUENCE](#)
[_SNAPSHOT](#)
[_TABLE](#)
[_TABLESPACE](#)
[_TRIGGER](#)
[_TYPE](#)
[_USER](#)
[_VIEW](#)
[_EXCEPT set operator](#)
[_FETCH statement](#)

functions

ACOS
ADD_MONTHS
ASCII
ASCIISTR
ASIN
ATAN
ATAN2
BFILENAME
BIN_TO_NUM
BITAND
CARDINALITY
CHARTOROWID
CHR
COALESCE
COLLECT
COMPOSE
CONCAT
CONVERT
CORR_K
COS
COSH
CV
DBTIMEZONE
DECODE
DECOMPOSE
DEPTH
DEREF
DUMP
EMPTY_BLOB()
EMPTY_CLOB()
EXISTSNODE
EXTRACT
EXTRACTVALUE
FIRST
FIRST_VALUE
FROM_TZ
GREATEST
GROUP_ID()
GROUPING
GROUPING_ID
HEXTORAW
INITCAP
INSTR
ITERATION_NUMBER
LAG
LAST
LAST_DAY
LAST_VALUE
LEAD
LEAST
LENGTH
LENGTHB
LNNVL
LOCALTIMESTAMP
LOG
LPAD
LTRIM
MAKE_REF

MEDIAN
MONTHS_BETWEEN
NANVL
NCHAR
NEW_TIME
NEXT_DAY
NLS_CHARSET_DECL_LEN
NLS_CHARSET_ID
NLS_CHARSET_NAME
NLS_INITCAP
NLS_LOWER
NLS_UPPER
NLSSORT
NTILE
NULLIF
NUMTODSINTERVAL
NUMTOYMINTERVAL
NVL
NVL2
ORA_HASH
PATH
POWERMULTISET
PRESENTNNV
PRESENTV
PREVIOUS
RATIO_TO_REPORT
RAWTOHEX
REF
REFTOHEX
REGEXP_INSTR
REGEXP_REPLACE
REGEXP_SUBSTR
REMAINDER
REPLACE
ROUND
ROWIDTOCHAR
RPAD
RTRIM
SCN_TO_TIMESTAMP
SESSIONTIMEZONE
SET
SIGN
SIN
SINH
SOUNDEX
STATS_BINOMIAL_TEST
STDDEV
STDDEV_SAMP
STDEV_POP
SUBSTR
SUBSTRB
SYS_CONNECT_BY_PATH
SYS_CONTEXT
SYS_DBURIGEN
SYS_EXTRACT_UTC
SYS_GUID()
SYS_TYPEID
SYS_XMLAGG
SYS_XMLGEN

_SYSDATE
_SYSTIMESTAMP
_TAN
_TANH
_TIMESTAMP_TO_SCN
_TO_BINARY_DOUBLE
_TO_BINARY_FLOAT
_TO_CHAR 2nd
_TO_CLOB
_TO_DATE
_TO_DSINTERVAL
_TO_LOB
_TO_MULTI_BYTE
_TO_NCHAR
_TO_NCLOB
_TO_NUMBER
_TO_SINGLE_BYTE
_TO_TIMESTAMP
_TO_TIMESTAMP_TZ
_TO_YMINTERVAL
_TRANSLATE 2nd
_TREAT
_TRUNC 2nd
_TZ_OFFSET
_UID
_UNISTR
_UPDATEXML
_USERENV
_VALUE
_VAR_POP
_VAR_SAMP
_VARIANCE
_VSIZE
_XMLAGG
_XMLCOLATTVAL
_XMLCONCAT
_XMLELEMENT
_XMLFOREST
_XMLSEQUENCE
_XMLTRANSFORM
_global temporary table
_GRANT statement
_IDENTIFIED BY
_object_privilege
_ON
_role
_system_privilege
_TO
_identifier size
_index-organized table (IOT)
_indexes
_INIT.ORA file
_INSERT statement
_ALL
_conditional multitable
_ELSE
_FIRST
_INSERT
_subquery

- [_TABLE](#)
- [_table_name](#)
- [_unconditional_multitable](#)
- [_VALUES](#)
- [_WHEN](#)
- [_INTERSECT_set_operator](#)
- [_JOIN_subclause](#)
- [_keywords](#)
- [_LIKE_operator](#)
- [_local_index_partitioning](#)
- [_logfiles](#)
- [_MERGE_statement](#)
- [_nested_tables](#)
- [_object-type_tables](#)
- [_OPEN_statement](#)
- [_ORDER_BY_clause](#)
- [_ORGANIZATION_HEAP_clause](#)
- [_partitioned_and_subpartitioned_tables](#)
 - [_altering](#)
- [_physical_attributes_clause](#)
- [_primary_databases](#)
- [_RELEASE_SAVEPOINT_statement](#)
- [_reserved_identifiers](#)
- [_RETURN_statement](#)
- [_REVOKE_statement](#)
- [_ROLLBACK_statement](#)
- [_SAVEPOINT_statement](#)
- [_schema_addressing](#)
- [_SELECT_statement](#)
 - [_\[CROSS\]_JOIN](#)
 - [_ALL](#)
 - [_AS](#)
 - [_DISTINCT](#)
 - [_flashback_queries](#)
 - [_flashback_query_clause](#)
 - [_FOR_UPDATE](#)
 - [_FROM](#)
 - [_FULL_\[OUTER\]_JOIN](#)
 - [_INNER_JOIN](#)
 - [_INTO](#)
 - [_JOIN](#)
 - [_LEFT_\[OUTER\]_JOIN](#)
 - [_MODEL_clause](#)
 - [_NATURAL](#)
 - [_ON](#)
 - [_optimizer_hint](#)
 - [_ORDER_SIBLINGS_BY](#)
 - [_PARTITION](#)
 - [_PARTITION_BY](#)
 - [_partitioned_outer_joins](#)
 - [_RIGHT_\[OUTER\]_JOIN](#)
 - [_SAMPLE](#)
 - [_select_item](#)
 - [_SUBPARTITION](#)
 - [_subquery](#)
 - [_TABLE](#)
 - [_UNIQUE](#)
 - [_USING](#)
 - [_VERSIONS_BETWEEN](#)

- [_WHERE](#)
- [_WITH query_name AS](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME_ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_special rules](#)
- [_standby databases](#)
- [_START TRANSACTION statement](#)
- [_storage_clause and LOBs](#)
- [_stored procedures](#)
- [_SUBQUERY substatement](#)
- [_syntax for object table](#)
- [_syntax for XMLType table](#)
- [_syntax to alter existing database](#)
- [_syntax to create new database](#)
- [_system privileges](#)
 - [_{ALTER | DROP} ANY ROLE](#)
 - [_{CREATE | ALTER | DROP | EXECUTE} ANY PROCEDURE](#)
 - [_{CREATE | ALTER | DROP | SELECT} ANY SEQUENCE](#)
 - [_{CREATE | ALTER | DROP} ANY CLUSTER](#)
 - [_{CREATE | ALTER | DROP} ANY DIMENSION](#)
 - [_{CREATE | ALTER | DROP} ANY INDEX](#)
 - [_{CREATE | ALTER | DROP} ANY MATERIALIZED VIEW](#)
 - [_{CREATE | ALTER | DROP} ANY SNAPSHOT](#)
 - [_{CREATE | ALTER | DROP} ANY TYPE](#)
 - [_{CREATE | ALTER | DROP} PROFILE](#)
 - [_{CREATE | ALTER | DROP} ROLLBACK SEGMENT](#)
 - [_{CREATE | ALTER | DROP} TABLESPACE](#)
 - [_{CREATE | DROP | ALTER} ANY OUTLINE](#)
 - [_{CREATE | DROP | EXECUTE} ANY INDEXTYPE](#)
 - [_{CREATE | DROP | EXECUTE} ANY OPERATOR](#)
 - [_{CREATE | DROP} ANY DIRECTORY](#)
 - [_{CREATE | DROP} ANY CONTEXT](#)
 - [_{CREATE | DROP} ANY LIBRARY](#)
 - [_{CREATE | DROP} ANY SYNONYM](#)
 - [_{CREATE | DROP} ANY TRIGGER](#)
 - [_{CREATE | DROP} ANY VIEW](#)
 - [_{CREATE | DROP} PUBLIC SYNONYM](#)
- [_ADMINISTER DATABASE TRIGGER](#)
- [_ALTER ANY TABLE](#)
- [_ALTER ANY TRIGGER](#)
- [_ALTER DATABASE](#)
- [_ALTER RESOURCE COST](#)
- [_ALTER SESSION](#)
- [_ALTER SYSTEM](#)
- [_ALTER USER](#)
- [_ANALYZE ANY](#)
- [_AUDIT ANY](#)
- [_AUDIT SYSTEM](#)
- [_BACKUP ANY TABLE](#)
- [_BECOME USER](#)
- [_COMMENT ANY TABLE](#)
- [_CREATE ANY TABLE](#)

[_CREATE CLUSTER](#)
[_CREATE DATABASE LINK](#)
[_CREATE DIMENSION](#)
[_CREATE INDEXTYPE](#)
[_CREATE MATERIALIZED VIEW](#)
[_CREATE OPERATOR](#)
[_CREATE PROCEDURE](#)
[_CREATE PUBLIC DATABASE LINK](#)
[_CREATE ROLE](#)
[_CREATE SEQUENCE](#)
[_CREATE SESSION](#)
[_CREATE SNAPSHOT](#)
[_CREATE SYNONYM](#)
[_CREATE TRIGGER](#)
[_CREATE TYPE](#)
[_CREATE USER](#)
[_CREATE VIEW](#)
[_DEBUG ANY PROCEDURE](#)
[_DEBUG CONNECT SESSION](#)
[_DELETE ANY TABLE](#)
[_DROP ANY TABLE](#)
[_DROP PUBLIC DATABASE LINK](#)
[_DROP USER](#)
[_EXECUTE ANY TYPE](#)
[_EXEMPT ACCESS POLICY](#)
[_FLASHBACK ANY TABLE](#)
[_FORCE ANY TRANSACTION](#)
[_FORCE TRANSACTION](#)
[_GLOBAL QUERY REWRITE](#)
[_GRANT ANY OBJECT PRIVILEGE](#)
[_GRANT ANY PRIVILEGE](#)
[_GRANT ANY ROLE](#)
[_INSERT ANY TABLE](#)
[_LOCK ANY TABLE](#)
[_MANAGE TABLESPACE](#)
[_ON COMMIT REFRESH](#)
[_QUERY REWRITE](#)
[_RESTRICTED SESSION](#)
[_RESUMABLE](#)
[_SELECT ANY DICTIONARY](#)
[_SELECT ANY OUTLINE](#)
[_SELECT ANY TABLE](#)
[_SYSDBA](#)
[_SYSOPER](#)
[_UNDER ANY TYPE](#)
[_UNDER ANY VIEW](#)
[_UNLIMITED TABLESPACE](#)
[_UPDATE ANY TABLE](#)
[_TRUNCATE TABLE statement](#)
[_UDFs](#)
[_UNION set operator](#)
[_UPDATE statement](#)
[_WHERE clause](#)
[_window syntax](#)
[_XMLType tables](#)
[Oracle keywords](#)
[ORD function \(MySQL\)](#)
[ORDER BY clause 2nd 3rd](#)
[_DB2](#)

[_keywords](#)
[_MySQL](#)
[_Oracle](#)
[_platform support for](#)
[_PostgreSQL](#)
[_programming tips](#)
[_quick reference](#)
[_rules](#)
[_SQL Server](#)
[_SQL2003 syntax](#)
[ordering clause](#)
[ORGANIZATION HEAP clause \(Oracle\)](#)
[OVERLAY function](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[parameter object types \(ADO.NET\)](#)

[parameters](#)

[_binding input](#)

[_bound](#) [See bound parameters]

[parametrized statements](#)

[parentheses](#)

[_nested and operator precedence](#)

[PARSENAME function](#)

[_SQL Server](#)

[partitioned tables](#)

[_Oracle](#)

[_altering](#)

[partitioning clause](#)

[PASSWORD function \(MySQL\)](#)

[PATH datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[PATH function](#)

[_Oracle](#)

[_PostgreSQL](#)

[PATINDEX function](#)

[_SQL Server](#)

[PCLOSE function \(PostgreSQL\)](#)

[PERCENT_RANK function 2nd 3rd](#)

[PERCENTILE_CONT function 2nd](#)

[PERCENTILE_DISC function 2nd](#)

[PERIOD_ADD function \(MySQL\)](#)

[PERIOD_DIFF function \(MySQL\)](#)

[permissions](#)

[PERMISSIONS function](#)

[_SQL Server](#)

[physical_attributes_clause \(Oracle\)](#)

[PI\(\) function](#)

[_MySQL](#)

[_PostgreSQL](#)

[_SQL Server](#)

[PL/pgSQL](#)

[PL/SQL](#)

[platform comparisons](#)

[POINT datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[POINT function \(PostgreSQL\)](#)

[POLYGON datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[POLYGON function \(PostgreSQL\)](#)

[POPEN function \(PostgreSQL\)](#)

[POSITION function](#)

[_MySQL](#)

[POSSTR function \(DB2\)](#)

[PostgreSQL](#)

[_allowed characters in identifiers](#)

[_beginning identifier character](#)

- [_CLOSE_CURSOR_statement](#)
- [_COMMIT_statement](#)
- [_CONNECT_statement](#)
- [_connecting_using_JDBC](#)
- [_CREATE/ALTER_DATABASE_statements](#)
- [_CREATE/ALTER_FUNCTION/PROCEDURE_statements](#)
- [_CREATE/ALTER_INDEX_statements](#)
- [_CREATE/ALTER_METHOD_statements](#)
- [_CREATE/ALTER_ROLE_statements](#)
- [_CREATE/ALTER_SCHEMA_statements](#)
- [_CREATE/ALTER_TABLE_statements](#)
- [_CREATE/ALTER_TRIGGER_statements](#)
- [_CREATE/ALTER_TYPE_statements](#)
- [_CREATE/ALTER_VIEW_statements](#)
- [_datatypes](#)
- [_DECLARE_CURSOR_command](#)
- [_DELETE_statement](#)
- [_DISCONNECT_statement](#)
- [_DROP_statements](#)
 - [_\[_PROCEDURAL_\]LANGUAGE](#)
 - [_AGGEGATE](#)
 - [_DATABASE](#)
 - [_FUNCTION](#)
 - [_GROUP](#)
 - [_INDEX](#)
 - [_OPERATOR](#)
 - [_RULE](#)
 - [_SEQUENCE](#)
 - [_TABLE](#)
 - [_TRIGGER](#)
 - [_TYPE](#)
 - [_USER](#)
 - [_VIEW](#)
- [_EXCEPT_set_operator](#)
- [_FETCH_statement](#)
- [_functions](#)
 - [_ABSTIME](#)
 - [_ACOS](#)
 - [_AGE](#)
 - [_AREA](#)
 - [_ASCII](#)
 - [_ASIN](#)
 - [_ATAN](#)
 - [_ATAN2](#)
 - [_BOX](#)
 - [_BROADCAST](#)
 - [_CBRT](#)
 - [_CENTER](#)
 - [_CHAR](#)
 - [_CHAR_LENGTH](#)
 - [_CIRCLE](#)
 - [_COALESCE](#)
 - [_COS](#)
 - [_COT_2nd](#)
 - [_DATE_PART](#)
 - [_DATE_TRUNC](#)
 - [_DEGREES](#)
 - [_DIAMETER](#)
 - [_FLOAT](#)

FLOAT4
HEIGHT
HOST
INITCAP
INTEGER
INTERVAL
ISCLOSED
ISFINITE
ISOPEN
LENGTH
LOG
LPAD
LSEG
LTRIM
MASKLEN
NETMASK
NETWORK
NPOINTS
NULLIF
PATH
PCLOSE
PI()
POINT
POLYGON
POPEN
POW
RADIANS
RADIUS
RELTIME
ROUND
RPAD
RTRIM
SET_MASKLEN
SIN
SUBSTR
SUBSTRING
TAN
TEXT
TIMESTAMP
TO_CHAR
TO_DATE
TO_NUMBER
TO_TIMESTAMP
TRANSLATE
TRUNC
VARCHAR
WIDTH
GRANT statement
identifier size
INSERT statement
INTERSECT set operator
JOIN subclause
keywords
LIKE operator
MERGE statement
OPEN statement
ORDER BY clause
RELEASE SAVEPOINT statement
reserved identifiers

- [_RETURN_statement](#)
- [_REVOKE_statement](#)
- [_ROLLBACK_statement](#)
- [_SAVEPOINT_statement](#)
- [_schema_addressing](#)
- [_SELECT_statement](#)
 - [_\[_CROSS_\]JOIN](#)
 - [_ALL](#)
 - [_AS](#)
 - [_FOR_UPDATE_OF](#)
 - [_FROM](#)
 - [_FULL\[_OUTER_\]JOIN](#)
 - [_GROUP_BY](#)
 - [_INNER_JOIN](#)
 - [_INTO](#)
 - [_LEFT\[_OUTER_\]JOIN](#)
 - [_NATURAL](#)
 - [_ON](#)
 - [_ORDER_BY](#)
 - [_RIGHT\[_OUTER_\]JOIN](#)
 - [_select_item](#)
- [_SET_CONNECTION_statement](#)
- [_SET_CONSTRAINT_statement](#)
- [_SET_PATH_statement](#)
- [_SET_ROLE_statement](#)
- [_SET_SCHEMA_statement](#)
- [_SET_SESSION_AUTHORIZATION_statement](#)
- [_SET_statement](#)
- [_SET_TIME_ZONE_statement](#)
- [_SET_TRANSACTION_statement](#)
- [_START_TRANSACTION_statement](#)
- [_SUBQUERY_substatement](#)
- [_TRUNCATE_TABLE_statement](#)
- [_UNION_set_operator](#)
- [_UPDATE_statement](#)
- [_web_site](#)
- [_WHERE_clause](#)

PostgreSQL keywords

pound symbol (#)

POW function

- [_MySQL](#)
- [_PostgreSQL](#)

POWER function

- [_MySQL](#)

POWERMULTISET function (Oracle)

prefixes

PreparedStatement set methods (JDBC)

PRESENTNNV function (Oracle)

PRESENTV function (Oracle)

PREVIOUS function (Oracle)

primary databases (Oracle)

PRIMARY_KEY constraints

PROC_ROLE function

procedural programming

PROCEDURE, quick reference

PROCESSADMIN (SQL Server system role)

programming databases

- [_examples](#)
- [_overview](#)





< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

QUARTER function

[_DB2](#)

[_MySQL](#)

[quoted_identifier_symbol](#)

[quoted_identifiers](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[RADIANS function](#)

[_DB2](#)

[_MySQL](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[RADIUS function \(PostgreSQL\)](#)

[RAISE_ERROR function \(DB2\)](#)

[RAND function](#)

[_DB2](#)

[_SQL_Server](#)

[RAND\(\) function \(MySQL\)](#)

[RANK function 2nd](#)

[RANK\(\) function](#)

[RATIO_TO_REPORT function \(Oracle\)](#)

[RAW datatype](#)

[_Oracle](#)

[_platform comparison](#)

[RAWTOHEX function \(Oracle\)](#)

[RDBMSs \(Relational Database Management System\)](#)

[ReadCommitted isolation level \(Transaction object \(ADO.NET\)\)](#)

[ReadUncommitted isolation level \(Transaction object \(ADO.NET\)\)](#)

[REAL datatype](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[REAL function \(DB2\)](#)

[REC2XML function \(DB2\)](#)

[records](#)

[REF function \(Oracle\)](#)

[REFTOHEX function \(Oracle\)](#)

[REGEXP_INSTR function \(Oracle\)](#)

[REGEXP_REPLACE function \(Oracle\)](#)

[REGEXP_SUBSTR function \(Oracle\)](#)

[REGR family of functions](#)

[REGR_AVGX function 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[REGR_AVGY function 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[REGR_COUNT function 2nd 3rd 4th 5th 6th 7th 8th](#)

[REGR_INTERCEPT function 2nd 3rd 4th 5th 6th 7th](#)

[REGR_R2 function 2nd 3rd 4th 5th 6th](#)

[REGR_SLOPE function 2nd 3rd 4th 5th](#)

[REGR_SXX function 2nd 3rd 4th](#)

[REGR_SXY function 2nd 3rd](#)

[REGR_SYY function 2nd](#)

[Relational Database Management Systems \(RDBMSs\)](#)

[relational databases, principles of](#)

[RELEASE SAVEPOINT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[RELEASE_LOCK function \(MySQL\)](#)

[RELTIME function \(PostgreSQL\)](#)

[REMAINDER function \(Oracle\)](#)

[REPEAT function](#)

- [_DB2](#)
- [_MySQL](#)

[RepeatableRead isolation level \(Transaction object \(ADO.NET\)\)](#)

[REPLACE function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_SQL Server](#)

[REPLICATE function](#)

- [_SQL Server](#)

[reserved words 2nd](#)

[resources, freeing](#)

[ResultSet methods \(JDBC\)](#)

- [_Get](#)

[RETURN statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[REVERSE function](#)

- [_MySQL](#)
- [_SQL Server](#)

[REVOKE statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[RIGHT \[OUTER\] JOIN](#)

[RIGHT function](#)

- [_DB2](#)
- [_MySQL](#)
- [_SQL Server](#)

[ROLE_CONTAIN function](#)

[ROLE_ID function](#)

[ROLE_NAME function](#)

[ROLLBACK statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[ROUND function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[row processing](#)

[ROW_NUMBER function](#)

[ROWCNT function](#)

[ROWCOUNT_BIG\(.\) function](#)

- [_SQL Server](#)

[ROWID datatype](#)

- [_Oracle](#)
- [_platform comparison](#)

[ROWIDTOCHAR function \(Oracle\)](#)

[rows, filtering and comparison operators](#)

[ROWVERSION datatype](#)

- [_platform comparison](#)
- [_SQL Server 2nd](#)

[RPAD function](#)

- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)

[RTRIM function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL Server](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[sampling](#)

[SAVEPOINT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[scalar functions 2nd](#)

[_built-in](#)

[_categories of](#)

[scalar subqueries](#)

[Scalar subquery in SELECT item list](#)

[Scalar subquery in WHERE/HAVING clause](#)

[schema addressing](#)

[schemas](#)

[SCN_TO_TIMESTAMP function \(Oracle\)](#)

[SEC_TO_TIME function \(MySQL\)](#)

[SECOND function](#)

[_DB2](#)

[_MySQL](#)

[SECURITYADMIN \(SQL Server system role\)](#)

[SELECT statement](#)

[_DB2](#)

[_examples](#)

[_GROUP BY clause](#)

[_HAVING clause](#)

[_keywords](#)

[_ALL](#)

[_AS](#)

[_DISTINCT](#)

[_FROM](#)

[_GROUP BY](#)

[_HAVING](#)

[_JOIN](#)

[_ORDER BY](#)

[_WHERE](#)

[_MySQL](#)

[_operators and 2nd](#)

[_Oracle](#)

[_ORDER BY clause](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[SEQUEL \(Structured English Query Language\)](#)

SERIAL datatype

- [_platform_comparison](#)
- [_PostgreSQL](#)

SERIAL4 datatype

- [_platform_comparison](#)
- [_PostgreSQL](#)

SERIAL8 datatype

- [_platform_comparison](#)
- [_PostgreSQL](#)

[Serializable isolation level \(Transaction object \(ADO.NET\)\)](#)

[SERVERADMIN \(SQL Server system role\)](#)

[SESSION_USER function](#)

[SESSION_USER\(\) function \(MySQL\)](#)

[SESSIONTIMEZONE function \(Oracle\)](#)

[SET CATALOG statement, quick reference](#)

[SET COLLATION statement, quick reference](#)

[SET CONNECTION statement](#)

- [_DB2](#)
- [_description](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET CONSTRAINT statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

SET datatype

- [_MySQL](#)
- [_platform_comparison](#)

[SET DESCRIPTOR statement, quick reference](#)

[SET function \(Oracle\)](#)

[set methods \(JDBC PreparedStatement\)](#)

[set operations](#)

[SET PATH statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET ROLE statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET SCHEMA statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET SESSION AUTHORIZATION statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET TIME_ZONE statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET TRANSACTION statement](#)

- [_DB2](#)
- [_keywords](#)

- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET_MASKLEN function \(PostgreSQL\)](#)

[SETUPADMIN \(SQL Server system role\)](#)

[SHA function \(MySQL\)](#)

[SHA1 function \(MySQL\)](#)

[shared keywords](#)

[SHOW_ROLE\(\) function](#)

[SHOW_SEC_SERVICES\(\) function](#)

SIGN function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_SQL Server](#)

SIN function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[Single parameter object type \(ADO.NET\)](#)

SINH function

- [_DB2](#)
- [_Oracle](#)

SMALLDATETIME datatype

- [_platform comparison](#)
- [_SQL Server 2nd](#)
- [_Sybase adaptive server](#)

SMALLINT datatype

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[SMALLINT function \(DB2\)](#)

SMALLMONEY datatype

- [_platform comparison](#)
- [_SQL Server](#)

SOME operator 2nd

- [_keywords](#)
- [_platform differences](#)
- [_platform support for](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL2003 syntax](#)

[SORTKEY function](#)

SOUNDEX function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_SQL Server](#)

[SPACE function](#)

[_DB2](#)

[_MySQL](#)

[_SQL_Server](#)

[SQL command reference, quick reference](#)

[SQL dialects](#)

[SQL history and implementations](#)

[SQL Server](#)

[_allowed_characters_in_identifiers](#)

[_basics](#)

[_beginning_identifier_character](#)

[_BIGINT_datatype](#)

[_BINARY_datatype](#)

[_CALL_statement](#)

[_CHAR\(\)_datatype](#)

[_CLOSE_CURSOR_statement](#)

[_COMMIT_statement](#)

[_CONNECT_statement](#)

[_connecting_using_JDBC](#)

[_CREATE/ALTER_DATABASE_statements](#)

[_ADD_FILEGROUP](#)

[_ADD_LOG](#)

[_ANSI_NULL_DEFAULT](#)

[_ANSI_NULLS](#)

[_ANSI_PADDING](#)

[_ANSI_WARNINGS](#)

[_ARITHABORT](#)

[_AUTO_CLOSE](#)

[_AUTO_CREATE_STATISTICS](#)

[_auto_option](#)

[_AUTO_SHRINK](#)

[_AUTO_UPDATE_STATISTICS](#)

[_COLLATE](#)

[_CONCAT_NULL_YIELDS_NULL](#)

[_CURSOR_CLOSE_ON_COMMIT](#)

[_CURSOR_DEFAULT](#)

[_FOR_LOAD](#)

[_MODIFY_FILE](#)

[_MODIFY_FILEGROUP](#)

[_MODIFY_NAME](#)

[_MULTI_USER](#)

[_NUMERIC_ROUNDABORT](#)

[_OFFLINE](#)

[_ON_file_definition](#)

[_QUOTED_IDENTIFIER](#)

[_READ_ONLY](#)

[_RECOVERY](#)

[_RECURSIVE_TRIGGERS](#)

[_REMOVE_FILE](#)

[_REMOVE_FILEGROUP](#)

[_RESTRICTED_USER](#)

[_SET](#)

[_SINGLE_USER](#)

[_sql_options_clause](#)

[_TORN_PAGE_DETECTION](#)

[_WITH](#)

[_CREATE/ALTER_FUNCTION/PROCEDURE_statements](#)

[_@parameter_datatype](#)

[_CREATE](#)

- ENCRYPTION
- FOR REPLICATION
- RECOMPILE
- RETURNS
- SCHEMABINDING
- WITH
- CREATE/ALTER INDEX statements
 - ASC
 - CLUSTERED
 - DESC
 - DROP_EXISTING
 - FILLFACTOR
 - IGNORE_DUP_KEY
 - ON
 - PAD_INDEX
 - SORT_IN_TEMPDB
 - STATISTICS_NORECOMPUTE
 - WITH
- CREATE/ALTER METHOD statements
- CREATE/ALTER ROLE statements
- CREATE/ALTER SCHEMA statements
- CREATE/ALTER TABLE statements
 - CHECK
 - CHECK CONSTRAINT
 - COLLATE
 - DEFAULT
 - ENABLE TRIGGER
 - FOREIGN KEY
 - IDENTITY
 - NOT FOR REPLICATION
 - PRIMARY KEY
 - ROWGUIDCOL
 - TEXTIMAGE_ON
 - WITH CHECK
- CREATE/ALTER TRIGGER statements
- CREATE/ALTER TYPE statements
- CREATE/ALTER VIEW statements
- datatypes
 - compared to Sybase adaptive server
- DATETIME datatype
- DECLARE CURSOR command
 - DYNAMIC
 - FAST_FORWARD
 - FOR
 - FOR UPDATE
 - FORWARD_ONLY
 - GLOBAL
 - INSENSITIVE
 - KEYSET
 - LOCAL
 - OPTIMISTIC
 - READ_ONLY
 - SCROLL
 - SCROLL_LOCKS
 - SQL-92 compatibility syntax
 - STATIC
 - Transact-SQL extension syntax
 - TYPE_WARNING
- DELETE statement

[delimiters and operators](#)

[DISCONNECT statement](#)

[DROP statements](#)

[DATABASE](#)

[DEFAULT](#)

[FUNCTION](#)

[INDEX](#)

[PROCEDURE](#)

[RULE](#)

[STATISTICS](#)

[TABLE](#)

[TRIGGER](#)

[TYPE](#)

[VIEW](#)

[EXCEPT set operator](#)

[FETCH statement](#)

[functions](#)

[ACOS](#)

[APP_NAME](#)

[ASCII](#)

[ASIN](#)

[ATAN](#)

[ATN2](#)

[BINARY_CHECKSUM](#)

[CHAR](#)

[CHARINDEX](#)

[CHECKSUM](#)

[CHECKSUM_AGG](#)

[COALESCE](#)

[COL_LENGTH](#)

[COL_NAME](#)

[CONTAINS](#)

[CONTAINSTABLE](#)

[CONVERT](#)

[COS](#)

[COT](#)

[DATABASEPROPERTYEX](#)

[DATALENGTH](#)

[DATEADD](#)

[DATEDIFF](#)

[DATENAME](#)

[DATEPART](#)

[DAY](#)

[DB_ID](#)

[DB_NAME](#)

[DEGREES](#)

[DIFFERENCE](#)

[FILE_ID](#)

[FILE_NAME](#)

[FILEGROUP_ID](#)

[FILEGROUP_NAME](#)

[FILEPROPERTY](#)

[FORMATMESSAGE](#)

[FREETEXT](#)

[FREETEXTTABLE](#)

[FULLTEXTCATALOGPROPERTY](#)

[FULLTEXTSERVICEPROPERTY](#)

[GETANSINULL](#)

[GETDATE\(.\)](#)

GETUTCDATE()
GROUPING
HOST_ID()
HOST_NAME()
IDENT_CURRENT
IDENT_INCR
IDENT_SEED
IDENTITY
INDEX_COL
INDEXPROPERTY
IS_MEMBER
IS_SRVROLEMEMBER
ISDATE
ISNULL
ISNUMERIC
LEFT
LEN
LOG
LOG10
LTRIM
MONTH
NCHAR
NEWID()
NULLIF
OBJECT_ID
OBJECT_NAME
OBJECTPROPERTY
OPEN
OPENDATASOURCE
OPENQUERY
OPENROWSET
PARSENAME
PATINDEX
PERMISSIONS
PI()
RADIANS
RAND
REPLACE
REPLICATE
REVERSE
RIGHT
ROUND
ROWCOUNT_BIG
RTRIM
SIGN
SIN
SOUNDEX
SPACE
STATS_DATE
STDEV
STDEVP
STR
STUFF
SUBSTRING
SUSER_ID
SUSER_SID
SUSER_SNAME
TAN
TEXTPTR

- [_TEXTVALID](#)
- [_TYPEPROPERTY](#)
- [_UNICODE](#)
- [_USER_ID](#)
- [_USER_NAME](#)
- [_VAR](#)
- [_VARP](#)
- [_YEAR](#)
- [_GRANT statement](#)
- [_identifier size](#)
- [_INSERT statement](#)
- [_INTERSECT set operator](#)
- [_JOIN subclause](#)
- [_keywords](#)
- [_LIKE operator](#)
- [_MERGE statement](#)
- [_NCHAR\(\) datatype 2nd](#)
- [_nested triggers](#)
- [_NTEXT datatype](#)
- [_OPEN statement](#)
- [_ORDER BY clause](#)
- [_recursive triggers](#)
- [_RELEASE SAVEPOINT statement](#)
- [_reserved identifiers](#)
- [_RETURN statement](#)
- [_REVOKE statement](#)
- [_ROLLBACK statement](#)
- [_ROWVERSION datatype](#)
- [_SAVEPOINT statement](#)
- [_schema addressing](#)
- [_SELECT statement](#)
 - [_COMPUTE clause](#)
 - [_FOR](#)
 - [_FROM](#)
 - [_GROUP BY](#)
 - [_INTO](#)
 - [_OPTION](#)
 - [_ORDER BY](#)
 - [_TOP](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_SMALLDATETIME datatype](#)
- [_SQL_VARIANT datatype](#)
- [_START TRANSACTION statement](#)
- [_stored procedures 2nd](#)
- [_SUBQUERY substatement](#)
- [_system roles](#)
 - [_BULKADMIN](#)
 - [_DB_ACCESSADMIN](#)
 - [_DB_BACKUPOPERATOR](#)
 - [_DB_DATAREADER](#)
 - [_DB_DATAWRITER](#)

- [_DB_DDLADMIN](#)
- [_DB_DENYDATAREADER](#)
- [_DB_DENYDATAWRITER](#)
- [_DB_OWNER](#)
- [_DB_SECURITYADMIN](#)
- [_DBCREATOR](#)
- [_DISKADMIN](#)
- [_PROCESSADMIN](#)
- [_SECURITYADMIN](#)
- [_SERVERADMIN](#)
- [_SETUPADMIN](#)
- [_SYSADMIN](#)
- [_TRUNCATE_TABLE statement](#)
- [_UDFs](#)
- [_UNION set operator](#)
- [_UNIQUEIDENTIFIER datatype](#)
- [_UPDATE statement](#)
- [_VARBINARY\(\) datatype](#)
- [_VARCHAR\(\) datatype](#)
- [_versus Sybase adaptive server](#)
 - [_functions](#)
- [_web site](#)
- [_WHERE clause](#)
- [SQL Standard web site](#)
- [SQL/CLI \(Call-Level Interface\)](#)
- [SQL/Foundation](#)
- [SQL/Framework](#)
- [SQL/JRT \(Java Routines and Types\)](#)
- [SQL/MED \(Management of External Data\)](#)
- [SQL/OBJ \(Object Language Binding\)](#)
- [SQL/PSM \(Persistent Stored Modules\)](#)
- [SQL/Schemata](#)
- [SQL/XML 2nd](#)
- [SQL_VARIANT datatype](#)
 - [_platform comparison](#)
 - [_SQL Server 2nd](#)
- [SQL2003](#)
 - [_allowed characters in identifiers](#)
 - [_beginning identifier character](#)
 - [_categories and datatypes](#)
 - [_data structures](#)
 - [_identifier size](#)
 - [_keywords](#)
 - [_new features](#)
 - [_numeric functions](#)
 - [_reserved identifiers](#)
 - [_schema addressing](#)
 - [_statement classes](#)
 - [_supplemental features packages](#)
 - [_window syntax](#)
- [SQL2003 keywords](#)
- [SQL2003 syntax](#)
 - [_ALL operator](#)
 - [_ANY operator](#)
 - [_BETWEEN operator](#)
 - [_CALL statement](#)
 - [_CLOSE CURSOR statement](#)
 - [_COMMIT statement](#)
 - [_CONNECT statement](#)

- [_CREATE/ALTER FUNCTION/PROCEDURE statements](#)
- [_CREATE/ALTER METHOD statements](#)
- [_CREATE/ALTER ROLE statements](#)
- [_CREATE/ALTER SCHEMA statements](#)
- [_CREATE/ALTER TABLE statements](#)
- [_CREATE/ALTER TRIGGER statements](#)
- [_CREATE/ALTER TYPE statements](#)
- [_CREATE/ALTER VIEW statements](#)
- [_DECLARE CURSOR command](#)
- [_DELETE statement](#)
- [_DISCONNECT statement](#)
- [_DROP statements](#)
- [_EXCEPT set operator](#)
- [_EXISTS operator](#)
- [_FETCH statement](#)
- [_GRANT statement](#)
- [_IN operator](#)
- [_INSERT statement](#)
- [_INTERSECT set operator](#)
- [_IS operator](#)
- [_JOIN subclause](#)
- [_LIKE operator](#)
- [_MERGE statement](#)
- [_OPEN statement](#)
- [_ORDER BY clause](#)
- [_RELEASE SAVEPOINT statement](#)
- [_RETURN statement](#)
- [_REVOKE statement](#)
- [_ROLLBACK statement](#)
- [_SAVEPOINT statement](#)
- [_SELECT statement](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_SOME operator](#)
- [_START TRANSACTION statement](#)
- [_SUBQUERY substatement](#)
- [_UNION set operator](#)
- [_UPDATE statement](#)
- [_WHERE clause](#)

SQL92

SQL99

- [_deleted elements in SQL2003](#)

SqlConnection

- [_connection string attributes for](#)

SQLPL

SQRT function

standby databases (Oracle)

START TRANSACTION statement

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)

- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)
- statement objects
 - [_associating SQL with](#)
 - [_creating](#)
 - [_executing](#)
- statements
 - [_executing](#)
 - [_preparing and executing](#)
 - [_re-executing](#)
 - [_Sybase adaptive server SQL](#)
- [STATS_BINOMIAL_TEST function \(Oracle\)](#)
- [STATS_DATE function](#)
 - [_SQL Server](#)
- [STD function \(MySQL\)](#)
- STDDEV function
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
- [STDDEV_POP function 2nd](#)
- [STDDEV_SAMP function 2nd](#)
 - [_Oracle](#)
- [STDEV function](#)
 - [_SQL Server](#)
- [STDEV_POP function \(Oracle\)](#)
- [STDEVVP function](#)
 - [_SQL Server](#)
- [STOP LOGICAL STANDBY APPLY](#)
- [storage_clause \(Oracle\)](#)
- stored procedures
 - [_operators and 2nd](#)
- [STR function \(SQL Server\)](#)
- [STRCMP function \(MySQL\)](#)
- string functions and operators
- [String parameter object type \(ADO.NET\)](#)
- [StringFixedLength parameter object type \(ADO.NET\)](#)
- [STUFF function \(SQL Server\)](#)
- [SUBDATE function \(MySQL\)](#)
- subpartitioned tables
 - [_Oracle](#)
 - [_altering](#)
- subqueries
 - [_correlated](#)
 - [_nested table](#)
 - [_scalar](#)
 - [_table](#)
- SUBQUERY substatement
 - [_DB2](#)
 - [_examples](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for 2nd](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_rules](#)

- [_SQL_Server](#)
- [_SQL2003_syntax](#)
- SUBSTR function
 - [_DB2](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
- SUBSTRB function (Oracle)
- SUBSTRING function
 - [_MySQL](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- SUBSTRING_INDEX function (MySQL)
- subtraction (-) arithmetic operator
- suffixes
- SUM function 2nd
- SUSER_ID function
 - [_SQL_Server](#)
- SUSER_NAME function
- SUSER_SID function
 - [_SQL_Server](#)
- SUSER_SNAME function
 - [_SQL_Server](#)
- SYB_SENDFMSG function
- Sybase Adaptive Aerver
 - [_TIMESTAMP datatype](#)
- Sybase Adaptive Server
 - [_BINARY datatype](#)
 - [_CHAR\(\) datatype](#)
 - [_datatypes](#)
 - [_compared to SQL_Server](#)
 - [_DATETIME datatype](#)
 - [_keywords 2nd](#)
 - [_NCHAR\(\) datatype](#)
 - [_NVARCHAR\(\) datatype](#)
 - [_SMALLDATETIME datatype](#)
 - [_SQL statements](#)
 - [_statements](#)
 - [_supported functions](#)
 - [_UNICHAR datatype](#)
 - [_UNIVARCHAR datatype](#)
 - [_VARBINARY\(\) datatype](#)
 - [_VARCHAR\(\) datatype](#)
 - [_versus SQL_Server](#)
 - [_functions](#)
- Sybase Adaptive Server keywords
- Sybase adaptive server keywords
- SYS_CONNECT_BY_PATH function (Oracle)
- SYS_CONTEXT function (Oracle)
- SYS_DBURIGEN function (Oracle)
- SYS_EXTRACT_UTC function (Oracle)
- SYS_GUID() function (Oracle)
- SYS_TYPEID function (Oracle)
- SYS_XMLAGG function (Oracle)
- SYS_XMLGEN function (Oracle)
- SYSADMIN (SQL Server system role)
- SYSDATE function (Oracle)
- SYSDATE() function (MySQL)
- system_delimiters
- SYSTEM_USER function

[SYSTEM_USER\(.\) function \(MySQL\)](#)

[SYSTIMESTAMP function \(Oracle\)](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

TABLE datatype

[_platform_comparison](#)
[_SQL_Server](#)

table_subqueries

[_nested](#)

table-level constraints

[TABLE_NAME function \(DB2\)](#)

[TABLE_SCHEMA function \(DB2\)](#)

tables

[TABLESAMPLE clause](#)

TAN function

[_DB2](#)
[_MySQL](#)
[_Oracle](#)
[_PostgreSQL](#)
[_SQL_Server](#)

TANH function

[_DB2](#)
[_Oracle](#)

[testing_expressions_against_range_of_values](#)

TEXT datatype

[_MySQL](#)
[_platform_comparison](#)
[_PostgreSQL](#)
[_SQL_Server](#)

[TEXT function \(PostgreSQL\)](#)

[TEXTPTR function \(SQL Server\)](#)

[TEXTVALID function \(SQL Server\)](#)

TIME datatype

[_DB2](#)
[_platform_comparison](#)
[_PostgreSQL](#)

[TIME function \(DB2\)](#)

[TIME_FORMAT function \(MySQL\)](#)

[TIME_TO_SEC function \(MySQL\)](#)

TIMESPAN datatype

[_platform_comparison](#)
[_PostgreSQL](#)

TIMESTAMP datatype

[_DB2](#)
[_Oracle](#)
[_platform_comparison](#)
[_PostgreSQL](#)
[_SQL_Server](#)
[_Sybase_adaptive_server](#)

TIMESTAMP function

[_DB2](#)
[_PostgreSQL](#)

[TIMESTAMP_FORMAT function \(DB2\)](#)

[TIMESTAMP_ISO function \(DB2\)](#)

[TIMESTAMP_TO_SCN function \(Oracle\)](#)

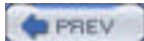
[TIMESTAMPDIFF function \(DB2\)](#)

TIMETZ datatype

[_platform_comparison](#)

- [_PostgreSQL](#)
- TINYINT datatype
 - [_platform_comparison](#)
 - [_SQL_Server](#)
- [TO_BINARY_DOUBLE function \(Oracle\)](#)
- [TO_BINARY_FLOAT function \(Oracle\)](#)
- TO_CHAR function
 - [_DB2](#)
 - [_Oracle_2nd](#)
 - [_PostgreSQL](#)
- [TO_CLOB function \(Oracle\)](#)
- TO_DATE function
 - [_DB2](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [TO_DAYS function \(MySQL\)](#)
- [TO_DSINTERVAL function \(Oracle\)](#)
- [TO_LOB function \(Oracle\)](#)
- [TO_MULTI_BYTE function \(Oracle\)](#)
- [TO_NCHAR function \(Oracle\)](#)
- [TO_NCLOB function \(Oracle\)](#)
- TO_NUMBER function
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [TO_SINGLE_BYTE function \(Oracle\)](#)
- TO_TIMESTAMP function
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [TO_TIMESTAMP_TZ function \(Oracle\)](#)
- [TO_UNICHAR function](#)
- [TO_YMINTERVAL function \(Oracle\)](#)
- Transact-SQL
- Transaction object
 - [_ADO.NET](#)
 - [_JDBC](#)
- [TRANSACTION_NONE isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_READ_COMMITTED isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_READ_UNCOMMITTED isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_REPEATABLE_READ isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_SERIALIZABLE isolation level \(Transaction object \(JDBC\)\)](#)
- transactions
 - [_beginning_2nd](#)
 - [_ADO.NET](#)
 - [_JDBC](#)
 - [_closing_and_making_permanent](#)
 - [_committing](#)
 - [_ending](#)
 - [_explicit](#)
 - [_implicit](#)
 - [_managing](#)
 - [_rolling_back](#)
- TRANSLATE function
 - [_DB2](#)
 - [_Oracle_2nd](#)
 - [_PostgreSQL](#)
- [TREAT function \(Oracle\)](#)
- [TRIM function](#)
- TRUNC function
 - [_DB2](#)

- [_Oracle 2nd](#)
- [_PostgreSQL](#)
- TRUNCATE function
 - [_DB2](#)
 - [_MySQL](#)
- TRUNCATE TABLE statement
 - [_DB2](#)
 - [_defacto standard syntax](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_rules](#)
 - [_SQL Server](#)
- TSEQUEL function
- Twelve Principles of Relational Databases
- TYPE_ID function (DB2)
- TYPE_NAME function (DB2)
- TYPE_SCHEMA function (DB2)
- TYPEPROPERTY function
 - [_SQL Server](#)
- TZ_OFFSET function (Oracle)





[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[UCASE function \(MySQL\)](#)

[UHIGHSURR function](#)

[UID function \(Oracle\)](#)

[UInt parameter object type \(ADO.NET\)](#)

[ULOWSURR function](#)

[unary operators 2nd](#)

[UNCOMPRESS function \(MySQL\)](#)

[UNHEX function \(MySQL\)](#)

[UNICHAR datatype \(Sybase adaptive server\)](#)

[UNICODE function](#)

[_SQL Server](#)

[UNION set operator](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[UNIQUE constraints](#)

[UNIQUEIDENTIFIER datatype](#)

[_platform comparison](#)

[_SQL Server 2nd](#)

[UNISTR function \(Oracle\)](#)

[UNIVARCHAR datatype \(Sybase adaptive server\)](#)

[UNIX_TIMESTAMP function \(MySQL\)](#)

[Unspecified isolation level \(Transaction object \(ADO.NET\)\)](#)

[UPDATE statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_operators and 2nd](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[UPDATEXML function \(Oracle\)](#)

[UPPER function](#)

[URLs](#)

[_MySQL](#)

[_open source](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

[UROWID datatype](#)

[_Oracle](#)

[_platform comparison](#)

[USCALAR function](#)

[user](#)

[USER\(.\) function \(MySQL\)](#)

[USER_ID function \(SQL Server\)](#)

[USER_NAME function \(SQL Server\)](#)

[USERENV function \(Oracle\)](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[VALID_NAME function](#)

[VALID_USER function](#)

[VALUE function](#)

[_DB2](#)

[_Oracle](#)

[VAR function](#)

[_DB2](#)

[_SQL Server](#)

[VAR_POP function 2nd](#)

[_Oracle](#)

[VAR_SAMP function 2nd](#)

[_Oracle](#)

[VARBINARY datatype](#)

[_platform comparison](#)

[_SQL Server](#)

[VARBINARY\(\) datatype](#)

[_SQL Server](#)

[_Sybase adaptive server](#)

[VARBIT datatype](#)

[_platform comparison](#)

[VARCHAR datatype](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[VARCHAR FOR BIT DATA datatype](#)

[_DB2](#)

[_platform comparison](#)

[VARCHAR function](#)

[_DB2](#)

[_PostgreSQL](#)

[VARCHAR\(\) datatype](#)

[_SQL Server](#)

[_Sybase adaptive server](#)

[VARCHAR_FORMAT function \(DB2\)](#)

[VARCHAR2 datatype](#)

[_Oracle](#)

[_platform comparison](#)

[VARGRAPHIC datatype](#)

[_DB2](#)

[_platform comparison](#)

[VARGRAPHIC function \(DB2\)](#)

[VARIANCE function](#)

[_DB2](#)

[_Oracle](#)

[VARP function](#)

[_SQL Server](#)

[Vector subquery in WHERE/HAVING clause](#)

[VERSION\(\) function \(MySQL\)](#)

[views 2nd](#)

[_operators and 2nd](#)

[VSIZE function \(Oracle\)](#)





[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Y\]](#)

WEEK function

[_DB2](#)

[_MySQL](#)

[WEEK_ISO function \(DB2\)](#)

[WEEKDAY function \(MySQL\)](#)

[WHERE clause 2nd 3rd](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

WHERE clause (SELECT statement)

[_comparison operators and](#)

[_logical operators and](#)

[WIDTH function \(PostgreSQL\)](#)

[WIDTH_BUCKET function](#)

[window functions 2nd](#)

[_DB2 syntax](#)

[_group clause](#)

[_Oracle syntax](#)

[_ordering clause](#)

[_partitioning clause](#)

[_SQL2003 syntax](#)

[windowing functions](#)

[WITH\[OUT\] SESSION SHUTDOWN](#)





[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[Xbase](#)

XML datatype

[_Oracle](#)

[_platform_comparison](#)

XML schema

[_tables_based_on](#)

[XMLAGG function \(Oracle\)](#)

[XMLCOLATTVAL function \(Oracle\)](#)

[XMLCONCAT function \(Oracle\)](#)

[XMLELEMENT function \(Oracle\)](#)

[XMLFOREST function \(Oracle\)](#)

[XMLSEQUENCE function \(Oracle\)](#)

[XMLTRANSFORM function \(Oracle\)](#)





< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[YEAR function](#)

[_DB2](#)

[_MySQL](#)

[_SQL Server](#)

[YEARWEEK function \(MySQL\)](#)



< Day Day Up >



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[ABS function](#)

[ABSTIME function \(PostgreSQL\)](#)

[ABSVAL function \(DB2\)](#)

[ACOS function](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

[ADD_MONTHS function \(Oracle\)](#)

[ADDDATE function \(MySQL\)](#)

[addition \(+\) arithmetic operator](#)

[ADO.NET](#)

[_beginning transactions](#)

[_bound parameters](#)

[_closing connection](#)

[_committing transactions](#)

[_database connection](#)

[_database program example](#)

[_DataReader methods](#)

[_Get](#)

[_error handling](#)

[_executing statements](#)

[_parameter object types](#)

[_retrieving data](#)

[_rolling back transactions](#)

[AGE function \(PostgreSQL\)](#)

[aggregate functions](#)

[ALL operator 2nd](#)

[_keywords](#)

[_platform differences](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[ALTER DATABASE statement](#) [See CREATE/ALTER DATABASE statements]

[ALTER DOMAIN statement, quick reference](#)

[ALTER FUNCTION statement](#) [See CREATE/ALTER FUNCTION/PROCEDURE statements]

[ALTER INDEX statement](#) [See CREATE/ALTER INDEX statements]

[ALTER METHOD statement](#) [See CREATE/ALTER METHOD statements]

[ALTER PROCEDURE statement](#) [See CREATE/ALTER FUNCTION/PROCEDURE statements]

[ALTER TABLE statement](#) [See CREATE/ALTER TABLE statements]

[ALTER TRIGGER statement](#) [See CREATE/ALTER TRIGGER statements]

[ALTER TYPE statement](#) [See CREATE/ALTER TYPE statements]

[ALTER VIEW statement](#) [See CREATE/ALTER VIEW statements]

[AND operator](#)

[ANSI standard](#)

[AnsiString parameter object type \(ADO.NET\)](#)

[AnsiStringFixedLength parameter object type \(ADO.NET\)](#)

[ANY operator 2nd](#)

[_keywords](#)

[_platform differences](#)

[_platform support for](#)

- [_programming_tips](#)
- [_quick_reference](#)
- [_rules](#)
- [_SQL2003_syntax](#)
- [APP_NAME function](#)
- [_SQL_Server](#)
- [AREA function \(PostgreSQL\)](#)
- [arithmetic operators 2nd](#)
- [AS keyword and assignment operator](#)
- [ASCII function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- [ASCIISTR function \(Oracle\)](#)
- [ASIN function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- [assignment \(=\) operator](#)
- [assignment operators](#)
- [ATAN function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- [ATAN2 function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [ATANH function \(DB2\)](#)
- [ATN2 function \(SQL Server\)](#)
- [AuthorizationID](#)
- [AVG function 2nd](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[BENCHMARK function \(MySQL\)](#)

[BETWEEN clause](#)

[BETWEEN operator 2nd](#)

[_keywords](#)

[_platform differences](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[BFILE datatype](#)

[_Oracle](#)

[_platform comparison](#)

[BFILENAME function \(Oracle\)](#)

[BIGINT datatype](#)

[_DB2](#)

[_MySQL](#)

[_platform comparison](#)

[_SQL Server 2nd](#)

[BIGINT function \(DB2\)](#)

[BIGSERIAL datatype](#)

[_platform comparison](#)

[_PostgreSQL 2nd](#)

[BIN function \(MySQL\)](#)

[BIN_TO_NUM function \(Oracle\)](#)

[BINARY datatype](#)

[_platform comparison](#)

[_SQL Server 2nd](#)

[_Sybase adaptive server](#)

[BINARY function \(MySQL\)](#)

[Binary parameter object type \(ADO.NET\)](#)

[BINARY_CHECKSUM function](#)

[_SQL Server](#)

[BINARY_DOUBLE datatype](#)

[_Oracle](#)

[_platform comparison](#)

[BINARY_FLOAT datatype](#)

[_Oracle](#)

[_platform comparison](#)

[BIT datatype](#)

[_MySQL](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[BIT VARYING datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[BIT_AND function \(MySQL\)](#)

[BIT_COUNT function \(MySQL\)](#)

[BIT_LENGTH function](#)

[BIT_OR function \(MySQL\)](#)

[BITAND function \(Oracle\)](#)

[bitwise AND \(&\) operator](#)

[bitwise exclusive OR \(^\) operator](#)

[bitwise operators](#)

[bitwise OR \(|\) operator](#)

BLOB datatype

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[BLOB function \(DB2\)](#)

BOOL datatype

[_MySQL](#)

[_platform comparison](#)

[_PostgreSQL](#)

[Boolean comparison operators](#)

BOOLEAN datatype

[_platform comparison](#)

[_PostgreSQL](#)

[Boolean parameter object type \(ADO.NET\)](#)

bound parameters

[_ADO.NET](#)

[_JDBC](#)

BOX datatype

[_platform comparison](#)

[_PostgreSQL](#)

[BOX function \(PostgreSQL\)](#)

[BROADCAST function \(PostgreSQL\)](#)

[built-in scalar functions](#)

[BULKADMIN \(SQL Server system role\)](#)

[Byte parameter object type \(ADO.NET\)](#)

BYTEA datatype

[_platform comparison](#)

[_PostgreSQL](#)



[SYMBOL] [A] [B] [**C**] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[CALL statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)
- [_versus EXECUTE statement](#)

[candidate key](#)

[CARDINALITY function \(Oracle\)](#)

[CASE function](#)

[CAST function 2nd](#)

[catalogs](#)

[CBRT function \(PostgreSQL\)](#)

[CEIL function](#)

[CENTER function \(PostgreSQL\)](#)

[Chaos isolation level \(Transaction object \(ADO.NET\)\)](#)

[CHAR datatype](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[CHAR FOR BIT DATA datatype](#)

- [_DB2](#)
- [_platform comparison](#)

[CHAR function](#)

- [_DB2](#)
- [_MySQL](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[CHAR VARYING datatype](#)

- [_DB2](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_SQL Server](#)

[CHAR\(\) datatype](#)

- [_SQL Server](#)
- [_Sybase adaptive server](#)

[CHAR_LENGTH function 2nd](#)

- [_PostgreSQL](#)

[CHARACTER datatype](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[character sets](#)

[character string literals](#)

CHARACTER VARYING datatype

[_DB2](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[CHARINDEX function \(SQL Server\)](#)

[CHARTOROWID function \(Oracle\)](#)

[CHECK constraints](#)

[CHECKSUM function](#)

[_SQL Server](#)

[CHECKSUM_AGG function](#)

[_SQL Server](#)

CHR function

[_DB2](#)

[_Oracle](#)

CIDR datatype

[_platform comparison](#)

[_PostgreSQL](#)

CIRCLE datatype

[_platform comparison](#)

[_PostgreSQL](#)

[CIRCLE function \(PostgreSQL\)](#)

CLOB datatype

[_DB2](#)

[_Oracle](#)

[_platform comparison](#)

[CLOB function \(DB2\)](#)

[CLOSE CURSOR statement](#)

[_DB2](#)

[_keywords](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

clusters

[COALESCE function](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

Codd, E. F.

[COL_LENGTH function \(SQL Server\)](#)

[COL_NAME function \(SQL Server\)](#)

collations

[COLLECT function \(Oracle\)](#)

column-level constraints

[column_constraint_clause \(Oracle\)](#)

columns

[COMMIT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

- [_platform_support_for](#)
- [_PostgreSQL](#)
- [_programming_tips](#)
- [_quick_reference](#)
- [_rules](#)
- [_SQL_Server](#)
- [_SQL2003_syntax](#)
- [COMMIT TO SWITCHOVER TO](#)
- [COMPARE function](#)
- [comparison_operators_2nd](#)
- [COMPOSE function \(Oracle\)](#)
- [COMPRESS function \(MySQL\)](#)
- [compressed_tables_in_Oracle](#)
- [CONCAT function](#)
 - [_DB2](#)
 - [_Oracle](#)
- [CONCAT_WS function \(MySQL\)](#)
- [concatenation_operator](#)
- [CONNECT statement](#)
 - [_DB2](#)
 - [_keywords](#)
 - [_Oracle](#)
 - [_platform_support_for](#)
 - [_PostgreSQL](#)
 - [_programming_tips](#)
 - [_quick_reference](#)
 - [_rules](#)
 - [_SQL_Server](#)
 - [_SQL2003_syntax](#)
- [CONNECTION_ID\(.\) function \(MySQL\)](#)
- [connections \(database\)](#)
 - [_closing](#)
 - [_opening](#)
 - [_ADO.NET](#)
 - [_JDBC](#)
- [connectivity_establishing](#)
- [constraints](#)
 - [_CHECK](#)
 - [_FOREIGN_KEY](#)
 - [_PRIMARY_KEY](#)
 - [_scope](#)
 - [_syntax](#)
 - [_UNIQUE](#)
- [CONTAINS function](#)
 - [_SQL_Server](#)
- [CONTAINSTABLE function](#)
 - [_SQL_Server](#)
- [CONV function \(MySQL\)](#)
- [CONVERT function](#)
 - [_Oracle](#)
 - [_SQL_Server](#)
- [CORR function_2nd](#)
- [CORR_K function \(Oracle\)](#)
- [correlated_subqueries](#)
- [Correlated_subquery_in_WHERE/HAVING_clause](#)
- [COS function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)

- [_PostgreSQL](#)
- [_SQL Server](#)
- COSH function
 - [_DB2](#)
 - [_Oracle](#)
- COT function
 - [_DB2](#)
 - [_MySQL](#)
 - [_PostgreSQL 2nd](#)
 - [_SQL Server](#)
- COUNT function 2nd
- COUNT_BIG function
- COVAR_POP function 2nd
- COVAR_SAMP function 2nd
- CREATE/ALTER DATABASE statements
 - [_DB2 \[See DB2, CREATE/ALTER DATABASE statements\]](#)
 - [_MySQL](#)
 - [_Oracle \[See Oracle, CREATE/ALTER DATABASE statements\]](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference 2nd](#)
 - [_rules](#)
 - [_SQL Server \[See SQL Server, CREATE/ALTER DATABASE statements\]](#)
- CREATE/ALTER DOMAIN statements, [quick reference](#)
- CREATE/ALTER FUNCTION/PROCEDURE statements
 - [_DB2 \[See DB2, CREATE/ALTER FUNCTION/PROCEDURE statements\]](#)
 - [_keywords](#)
 - [_CALL ON NULL INPUT](#)
 - [_CASCADE](#)
 - [_code_block](#)
 - [_CONTAINS SQL](#)
 - [_CREATE](#)
 - [_DETERMINISTIC](#)
 - [_DYNAMIC RESULT SETS](#)
 - [_IN](#)
 - [_LANGUAGE](#)
 - [_MODIFIES SQL DATA](#)
 - [_NAME](#)
 - [_NO SQL](#)
 - [_OUT](#)
 - [_PARAMETER STYLE](#)
 - [_READS SQL DATA](#)
 - [_RESTRICT](#)
 - [_RETURN NULL ON NULL INPUT](#)
 - [_RETURNS](#)
 - [_SPECIFIC](#)
 - [_STATIC DISPATCH](#)
 - [_MySQL](#)
 - [_Oracle \[See Oracle, CREATE/ALTER FUNCTION/PROCEDURE statements\]](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference 2nd](#)
 - [_rules](#)
 - [_SQL Server \[See SQL Server, CREATE/ALTER FUNCTION/PROCEDURE statements\]](#)
 - [_SQL2003 syntax](#)
- CREATE/ALTER INDEX statements
 - [_common vendor syntax](#)

- [_DB2](#) [See DB2, CREATE/ALTER INDEX statements]
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#) [See Oracle, CREATE/ALTER INDEX statements]
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference 2nd](#)
- [_rules](#)
- [_SQL Server](#) [See SQL Server, CREATE/ALTER INDEX statements]

[CREATE/ALTER METHOD statements](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference 2nd](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[CREATE/ALTER ROLE statements](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[CREATE/ALTER SCHEMA statements](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[CREATE/ALTER TABLE statements](#)

- [_DB2](#) [See DB2, CREATE/ALTER TABLE statements]
- [_keywords](#)
- [_ADD 2nd](#)
- [_ALTER](#)
- [_CASCADE](#)
- [_column_name](#)
- [_CONSTRAINT](#)
- [_DROP COLUMN](#)
- [_DROP CONSTRAINT](#)
- [_GLOBAL TEMPORARY](#)
- [_LIKE](#)
- [_LOCAL TEMPORARY](#)

[_OF](#)

[_ON COMMIT](#)

[_REF IS](#)

[_RESTRICT](#)

[_MySQL](#) [See MySQL, CREATE/ALTER TABLE statements]

[_Oracle](#) [See Oracle, CREATE/ALTER TABLE statements]

[_platform support for](#)

[_PostgreSQL](#) [See PostgreSQL, CREATE/ALTER TABLE statements]

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#) [See SQL Server, CREATE/ALTER TABLE statements]

[_SQL2003 syntax](#)

[CREATE/ALTER TRIGGER statements](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#) [See Oracle, CREATE/ALTER TRIGGER statements]

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference 2nd](#)

[_rules](#)

[_SQL Server \(SQL Server, CREATE/ALTER TRIGGER statements\)](#)

[_SQL2003 syntax](#)

[CREATE/ALTER TYPE statements](#)

[_DB2](#) [See DB2, CREATE/ALTER TYPE statements]

[_keywords](#)

[_MySQL](#)

[_Oracle](#) [See Oracle, CREATE/ALTER TYPE statements]

[_platform support for](#)

[_PostgreSQL](#) [See PostgreSQL, CREATE/ALTER TYPE statements]

[_programming tips](#)

[_quick reference 2nd](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[CREATE/ALTER VIEW statements](#)

[_DB2](#) [See DB2, CREATE/ALTER VIEW statements]

[_keywords](#)

[_MySQL](#)

[_Oracle](#) [See Oracle, CREATE/ALTER VIEW statements]

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference 2nd](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[CROSS JOIN](#)

[CUME_DIST function 2nd 3rd](#)

[CURDATE\(.\) function \(MySQL\)](#)

[Currency parameter object type \(ADO.NET\)](#)

[CURSOR datatype](#)

[_platform comparison](#)

[_SQL Server](#)

[cursors](#) [See CLOSE CURSOR statement DECLARE CURSOR command]

[CURTIME\(.\) function \(MySQL\)](#)

[CV function \(Oracle\)](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

data

[_retrieving](#)

[_ADO.NET](#)

[_JDBC](#)

[Data Control Language \(DCL\)](#)

[Data Definition Language \(DDL\)](#)

[Data Manipulation Language \(DML\)](#)

[data structures](#)

[database objects and operators 2nd](#)

[database programming](#) [See programming databases]

[DATABASE\(\) function \(MySQL\)](#)

[DATABASEPROPERTYEX function](#)

[_SQL Server](#)

[DATALENGTH function \(SQL Server\)](#)

[DATALINK datatype](#)

[_DB2](#)

[_platform comparison](#)

[DataReader methods \(ADO.NET\)](#)

[_Get](#)

[datatypes](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

[_Sybase adaptive server](#)

[DATE datatype](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[DATE function \(DB2\)](#)

[DATE_FORMAT function \(MySQL\)](#)

[DATE_ADD function \(MySQL\)](#)

[DATE_PART function \(PostgreSQL\)](#)

[DATE_SUB function \(MySQL\)](#)

[DATE_TRUNC function \(PostgreSQL\)](#)

[DATEADD function \(SQL Server\)](#)

[DATEDIFF function \(SQL Server\)](#)

[DATENAME function \(SQL Server\)](#)

[DATEPART function \(SQL Server\)](#)

[DATETIME datatype](#)

[_MySQL](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server 2nd](#)

[_Sybase adaptive server](#)

[DateTime parameter object type \(ADO.NET\)](#)

[DAY function](#)

[_DB2](#)

[_SQL Server](#)

[DAYNAME function](#)

[_DB2](#)

MySQL

DAYOFMONTH function (MySQL)

DAYOFWEEK function

DB2

MySQL

DAYOFWEEK_ISO function (DB2)

DAYOFYEAR function

DB2

MySQL

DAYS function (DB2)

DB_ACCESSADMIN (SQL Server system role)

DB_BACKUPOPERATOR (SQL Server system role)

DB_DATAREADER (SQL Server system role)

DB_DATAWRITER (SQL Server system role)

DB_DDLADMIN (SQL Server system role)

DB_DENYDATAREADER (SQL Server system role)

DB_DENYDATAWRITER (SQL Server system role)

DB_ID function (SQL Server)

DB_NAME function (SQL Server)

DB_OWNER (SQL Server system role)

DB_SECURITYADMIN (SQL Server system role)

DB2

_allowed_characters_in_identifiers

_beginning_identifier_character

_CALL statement

_CLOSE CURSOR statement

_COMMIT statement

_CONNECT statement

_connecting_using JDBC

_CREATE/ALTER DATABASE statements

_ALIAS

_APPLY

_AT DBPARTITIONNUM

_AUTOCONFIGURE

_CATALOG TABLESPACE

_COLLATE USING

_CREATE {DATABASE | DB}

_DFT_EXTENT_SZ

_EXTENTSIZE

_MANAGED BY

_NUMSEGS

_ON

_OVERHEAD

_PREFETCHSIZE

_tablespace_definition

_TEMPORARY TABLESPACE

_TRANSFERRATE

_USER TABLESPACE

_USING CODESET

_WITH

_CREATE/ALTER FUNCTION/PROCEDURE statements

_CARDINALITY

_CONTAINS SQL

_DBINFO

_EXTERNAL NAME

_FENCED

_FINAL CALL

_INHERIT SPECIAL REGISTERS

_LANGUAGE

MODIFIES SQL DATA
NO SQL
PARALLEL
PARAMETER STYLE
PREDICATES
PROGRAM TYPE
READS SQL DATA
RETURNS
SCRATCHPAD
THREADSAFE
TRANSFORM GROUP
CREATE/ALTER INDEX statements
ALLOW REVERSE SCANS
ASC
CLUSTER
COLLECT STATISTICS
DESC
EXTEND USING
INCLUDE
MINPCTUSED
PCTFREE
SPECIFICATION ONLY
CREATE/ALTER METHOD statements
CREATE/ALTER ROLE statements
CREATE/ALTER SCHEMA statements
CREATE/ALTER TABLE statements
ACTIVATE NOT LOGGED INITIALLY
ACTIVATE VALUE COMPRESSION
ADD
ALTER [COLUMN]
ALTER {CHECK | FOREIGN KEY}
APPEND
AS
AS IDENTITY
COMPACT
COMPRESS SYSTEM DEFAULT
copy_options
DATA CAPTURE 2nd
DEFINITION ONLY
derived_table_options
DROP
FILE LINK CONTROL
GENERATED
IN
intermediate table, defining
LIKE
LINKTYPE URL
LOCKSIZE
LOGGED
MODE DB2OPTIONS
NLINE LENGTH
NO LINK CONTROL
NOT LOGGED INITIALLY
OPTIONS
ORGANIZE BY
PARTITIONING KEY
PCTFREE
REPLICATED
SCOPE

- [_SET MATERIALIZED_QUERY](#)
- [_UNDER](#)
- [_VALUE COMPRESSION](#)
- [_VOLATILE](#)
- [_WITH RESTRICT ON DROP](#)
- [CREATE/ALTER TRIGGER statements](#)
- [CREATE/ALTER TYPE statements](#)
 - [_ADD METHOD](#)
 - [_ALTER METHOD](#)
 - [_DISTINCT](#)
 - [_DROP METHOD](#)
 - [_FENCED](#)
 - [_INLINE LENGTH](#)
 - [_METHOD](#)
 - [_MODE DB2SQL](#)
 - [_OVERRIDING](#)
 - [_RESTRICT](#)
 - [_THREADSAFE](#)
 - [_WITH COMPARISONS](#)
 - [_WITHOUT COMPARISONS](#)
- [CREATE/ALTER VIEW statements](#)
 - [_WITH CHECK OPTION clause](#)
- [datalink-related attributes](#)
- [datatypes](#)
- [DECLARE CURSOR command](#)
- [DELETE statement](#)
- [DISCONNECT statement](#)
- [DROP statements](#)
 - [_\[DISTINCT\] TYPE](#)
 - [_\[SPECIFIC\] FUNCTION](#)
 - [_\[SPECIFIC\] METHOD](#)
 - [_BUFFERPOOL](#)
 - [_DATABASE PARTITION GROUP](#)
 - [_EVENT MONITOR](#)
 - [_FUNCTION MAPPING](#)
 - [_INDEX](#)
 - [_NICKNAME](#)
 - [_PACKAGE](#)
 - [_RESTRICT](#)
 - [_SCHEMA](#)
 - [_SEQUENCE](#)
 - [_SERVER](#)
 - [_TABLE](#)
 - [_TABLESPACE](#)
 - [_TRANSFORM](#)
 - [_TRIGGER](#)
 - [_TYPE MAPPING](#)
 - [_USER MAPPING FOR](#)
 - [_VIEW](#)
 - [_WRAPPER](#)
- [EXCEPT set operator](#)
- [external scalar user-defined functions](#)
- [external stored procedure](#)
- [external table user-defined functions](#)
- [FETCH statement](#)
- [functions](#)
 - [_ABSVAL](#)
 - [_ACOS](#)
 - [_ASCII](#)

ASIN
ATAN
ATAN2
ATANH
BIGINT
BLOB
CHAR
CHR
CLOB
COALESCE
CONCAT
COS
COSH
COT
DATE
DAY
DAYNAME
DAYOFWEEK
DAYOFWEEK_ISO
DAYOFYEAR
DAYS
DBCLOB
DBPARTITIONNUM
DEC
DECIMAL
DECRYPT_BIN
DECRYPT_CHAR
DEGREES
DEREF
DIFFERENCE
DIGITS
DOUBLE
ENCRYPT
EVENT_MON_STATE
FLOAT
GENERATE_UNIQUE()
GETHINT
GRAPHIC
HASHEDVALUE
HEX
HOUR
IDENTITY_VAL_LOCAL()
INSERT
INTEGER
JULIAN_DAY
LEFT
LENGTH
LOCATE
LOG
LOG10
LONG_VARCHAR
LONG_VARGRAPHIC
LTRIM
MICROSECOND
MIDNIGHT_SECONDS
MINUTE
MONTH
MONTHNAME
MULTIPLY_ALT

NULLIF
POSSTR
QUARTER
RADIANS
RAISE_ERROR
RAND
REAL
REC2XML
REPEAT
REPLACE
RIGHT
ROUND
RTRIM
SECOND
SIGN
SIN
SINH
SMALLINT
SOUNDEX
SPACE
STDDEV
SUBSTR
TABLE_NAME
TABLE_SCHEMA
TAN
TANH
TIME
TIMESTAMP
TIMESTAMP_FORMAT
TIMESTAMP_ISO
TIMESTAMPDIFF
TO_CHAR
TO_DATE
TRANSLATE
TRUNC
TRUNCATE
TYPE_ID
TYPE_NAME
TYPE_SCHEMA
VALUE
VAR
VARCHAR
VARCHAR_FORMAT
VARGRAPHIC
VARIANCE
WEEK
WEEK_ISO
YEAR

GRANT statement
{ALTER | USAGE} ON SEQUENCE
{ALTERIN | CREATEIN | DROPIN} ON SCHEMA
{BIND | CONTROL | EXECUTE} ON PACKAGE
CONTROL ON INDEX
database_privileges
EXECUTE ON
object_privilege
PASSTHRU ON SERVER
TO USER
USE OF TABLESPACE

- [_WITH GRANT OPTION](#)
- [_identifier size](#)
- [INSERT statement](#)
- [INTERSECT set operator](#)
- [JOIN subclause](#)
- [keywords](#)
- [LDAP](#)
- [LIKE operator](#)
- [MERGE statement](#)
- [OLE-DB external table user-defined functions](#)
- [OPEN statement](#)
- [ORDER BY clause](#)
- [RELEASE SAVEPOINT statement](#)
- [reserved identifiers](#)
- [RETURN statement](#)
- [REVOKE statement](#)
 - [_ { ALTERIN | CREATEIN | DROPIN } ON SCHEMA](#)
 - [_ { BIND | CONTROL | EXECUTE } ON PACKAGE](#)
 - [_ { ALTER | USAGE } ON SEQUENCE](#)
 - [_CONTROL ON INDEX](#)
 - [_database_privileges](#)
 - [_EXECUTE ON](#)
 - [_FROM](#)
 - [_levels of system privileges](#)
 - [_object_privilege](#)
 - [_PASSTHRU ON SERVER](#)
 - [_RESTRICT](#)
 - [_USE OF TABLESPACE](#)
- [ROLLBACK statement](#)
- [SAVEPOINT statement](#)
- [schema addressing](#)
- [SELECT statement](#)
- [SET CONNECTION statement](#)
- [SET CONSTRAINT statement](#)
- [SET PATH statement](#)
- [SET ROLE statement](#)
- [SET SCHEMA statement](#)
- [SET SESSION AUTHORIZATION statement](#)
- [SET statement](#)
- [SET TIME ZONE statement](#)
- [SET TRANSACTION statement](#)
- [sourced user-defined functions](#)
- [SQL stored procedure](#)
- [SQL user-defined functions](#)
- [START TRANSACTION statement](#)
- [SUBQUERY substatement](#)
- [TRUNCATE TABLE statement](#)
- [UNION set operator](#)
- [UPDATE statement](#)
- [web site](#)
- [WHERE clause](#)
- [window syntax](#)

DB2 keywords

DBCLOB datatype

- [_DB2](#)
- [_platform comparison](#)

DBCLOB function (DB2)

DBCREATOR (SQL Server system role)

DBPARTITIONNUM function (DB2)

[DBTIMEZONE function \(Oracle\)](#)

DEC datatype

[_DB2](#)

[_platform_comparison](#)

[_SQL_Server](#)

[DEC function \(DB2\)](#)

DECIMAL datatype

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform_comparison](#)

[_PostgreSQL_2nd](#)

[_SQL_Server](#)

[DECIMAL function \(DB2\)](#)

[Decimal parameter object type \(ADO.NET\)](#)

[declarative processing](#)

[DECLARE CURSOR command](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

DECODE function

[_MySQL](#)

[_Oracle](#)

[DECOMPOSE function \(Oracle\)](#)

[DECRYPT_BIN function \(DB2\)](#)

[DECRYPT_CHAR function \(DB2\)](#)

DEGREES function

[_DB2](#)

[_MySQL](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[DELETE statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_operators_and_2nd](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

[delimited identifiers](#)

[delimiters](#)

[DENSE_RANK function 2nd_3rd](#)

[DEPTH function \(Oracle\)](#)

DEREF function

[_DB2](#)

[_Oracle](#)

- [deterministic functions](#)
- [dialects, SQL](#)
- [DIAMETER function \(PostgreSQL\)](#)
- [DIFFERENCE function](#)
 - [_DB2](#)
 - [_SQL Server](#)
- [DIGITS function \(DB2\)](#)
- [DISCONNECT statement](#)
 - [_DB2](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference](#)
 - [_rules](#)
 - [_SQL Server](#)
 - [_SQL2003 syntax](#)
- [DISKADMIN \(SQL Server system role\)](#)
- [division \(/\) arithmetic operator](#)
- [DOUBLE datatype](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_platform comparison](#)
- [DOUBLE function \(DB2\)](#)
- [Double parameter object type \(ADO.NET\)](#)
- [DOUBLE PRECISION datatype](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform comparison](#)
 - [_PostgreSQL](#)
 - [_SQL Server](#)
- [double-pound symbol \(##\)](#)
- [DROP DATABASE statement, quick reference](#)
- [DROP DOMAIN statement, quick reference](#)
- [DROP FUNCTION statement, quick reference](#)
- [DROP INDEX statement, quick reference](#)
- [DROP METHOD statement, quick reference](#)
- [DROP ROLE statement, quick reference](#)
- [DROP statements](#)
 - [_DB2](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_quick reference](#)
 - [_rules](#)
 - [_SQL Server](#)
 - [_SQL2003 syntax](#)
- [DROP TABLE statement, quick reference](#)
- [DROP TRIGGER statement, quick reference](#)
- [DROP TYPE statement, quick reference](#)
- [DROP VIEW statement, quick reference](#)
- [DUMP function \(Oracle\)](#)



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[ELT function \(MySQL\)](#)

[EMPTY_BLOB\(.\) function \(Oracle\)](#)

[EMPTY_CLOB\(.\) function \(Oracle\)](#)

[ENCODE function \(MySQL\)](#)

ENCRYPT function

[_DB2](#)

[_MySQL](#)

ENUM datatype

[_MySQL](#)

[_platform comparison](#)

[equal to \(=\) comparison operator](#)

error handling

[_ADO.NET](#)

[_JDBC](#)

[EVENT_MON_STATE function \(DB2\)](#)

EXCEPT set operator

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[EXECUTE statement versus CALL statement](#)

EXISTS operator 2nd

[_platform differences](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[EXISTSNODE function \(Oracle\)](#)

EXP function

explicit transactions

[EXPORT_SET function \(MySQL\)](#)

expressions

[_operator precedence and](#)

[_testing against range of values](#)

EXTRACT function

[_Oracle](#)

[EXTRACTVALUE function \(Oracle\)](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[FETCH statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[FIELD function \(MySQL\)](#)

[FILE_ID function](#)

- [_SQL Server](#)

[FILE_NAME function](#)

- [_SQL Server](#)

[FILEGROUP_ID function](#)

- [_SQL Server](#)

[FILEGROUP_NAME function](#)

- [_SQL Server](#)

[FILEGROUPPROPERTY function](#)

- [_SQL Server](#)

[FILEPROPERTY function](#)

- [_SQL Server](#)

[filtering checks](#)

[FIND_IN_SET function \(MySQL\)](#)

[FIRST function \(Oracle\)](#)

[FIRST_VALUE function \(Oracle\)](#)

[FLOAT datatype](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_SQL Server 2nd](#)

[FLOAT function](#)

- [_DB2](#)
- [_PostgreSQL](#)

[FLOAT4 datatype](#)

- [_platform comparison](#)
- [_PostgreSQL](#)

[FLOAT4 function \(PostgreSQL\)](#)

[FLOAT8 datatype](#)

- [_platform comparison](#)
- [_PostgreSQL](#)

[FLOOR function](#)

[FOREIGN KEY constraints](#)

- [_CASCADE keyword](#)
- [_FOREIGN KEY keyword](#)
- [_MATCH keyword](#)
- [_NO ACTION keyword](#)
- [_ON DELETE keyword](#)
- [_ON UPDATE keyword](#)
- [_REFERENCES keyword](#)

- [_RESTRICT keyword](#)
- [_SET_DEFAULT keyword](#)
- [_SET_NULL keyword](#)
- [FORMAT function \(MySQL\)](#)
- [FORMATMESSAGE function](#)
 - [_SQL Server](#)
- [freeing_resources](#)
- [FREETEXT function \(SQL Server\)](#)
- [FREETEXTTABLE function](#)
 - [_SQL Server](#)
- [FROM clause](#)
- [FROM_DAYS function \(MySQL\)](#)
- [FROM_TZ function \(Oracle\)](#)
- [FROM_UNIXTIME function \(MySQL\) 2nd](#)
- [FULL \[OUTER\] JOIN](#)
- [FULLTEXTCATALOGPROPERTY function](#)
 - [_SQL Server](#)
- [FULLTEXTSERVICEPROPERTY function](#)
 - [_SQL Server](#)
- [functions](#)
 - [_ABS](#)
 - [_aggregate](#)
 - [_APP_NAME\(.\)](#)
 - [_AVG 2nd](#)
 - [_BINARY_CHECKSUM](#)
 - [_BIT_LENGTH](#)
 - [_CASE](#)
 - [_CAST 2nd](#)
 - [_CEIL](#)
 - [_CHAR_LENGTH 2nd](#)
 - [_CHECKSUM](#)
 - [_CHECKSUM_AGG](#)
 - [_COALESCE](#)
 - [_COMPARE](#)
 - [_CONTAINS](#)
 - [_CONTAINSTABLE](#)
 - [_CONVERT](#)
 - [_CORR 2nd](#)
 - [_COUNT 2nd](#)
 - [_COUNT_BIG](#)
 - [_COVAR_POP 2nd](#)
 - [_COVAR_SAMP 2nd](#)
 - [_CUME_DIST 2nd](#)
 - [_CUME_DIST\(.\)](#)
 - [_DATABASEPROPERTYEX](#)
 - [_DAY](#)
 - [_DB2 supported \[See DB2, functions\]](#)
 - [_DENSE_RANK 2nd](#)
 - [_DENSE_RANK\(.\)](#)
 - [_deterministic](#)
 - [_EXP](#)
 - [_EXTRACT](#)
 - [_FILE_ID](#)
 - [_FILE_NAME](#)
 - [_FILEGROUP_ID](#)
 - [_FILEGROUP_NAME](#)
 - [_FILEGROUPPROPERTY](#)
 - [_FILEPROPERTY](#)
 - [_FLOOR](#)

[_FORMATMESSAGE](#)
[_FREETEXTTABLE](#)
[_FULLTEXTCATALOGPROPERTY](#)
[_FULLTEXTSERVICEPROPERTY](#)
[_GETANSINULL](#)
[_GETUTCDATE\(\)](#)
[_GROUPING](#)
[_HEXTOINT](#)
[_IDENT_CURRENT](#)
[_IDENT_INCR](#)
[_IDENT_SEED](#)
[_IDENTITY](#)
[_INDEXPROPERTY](#)
[_INTTOHEX](#)
[_IS_MEMBER](#)
[_IS_SEC_SERVICE_ON](#)
[_IS_SRVROLEMEMBER](#)
[_ISDATE](#)
[_ISNUMERIC](#)
[_lct_admin](#)
[_LEFT](#)
[_LEN](#)
[_LICENSE_ENABLED](#)
[_LN](#)
[_LOWER](#)
[_MAX 2nd](#)
[_MIN 2nd](#)
[_MOD](#)
[_MONTH](#)
[_MUT_EXCL_ROLES](#)
[_MySQL supported \[See MySQL, functions\]](#)
[_NCHAR](#)
[_NEWID\(\)](#)
[_nondeterministic](#)
[_NULLIF](#)
[_numeric](#)
[_OBJECTPROPERTY](#)
[_OCTET_LENGTH](#)
[_OPEN](#)
[_OPENDATASOURCE](#)
[_OPENQUERY](#)
[_OPENROWSET](#)
[_operators and 2nd](#)
[_Oracle supported \[See Oracle, functions\]](#)
[_OVERLAY](#)
[_PARSENAME](#)
[_PATINDEX](#)
[_PERCENT_RANK 2nd 3rd](#)
[_PERCENTILE_CONT 2nd](#)
[_PERCENTILE_DISC 2nd](#)
[_PERMISSIONS](#)
[_POSITION](#)
[_PostgreSQL \[See PostgreSQL, functions\]](#)
[_POWER](#)
[_PROC_ROLE](#)
[_RANK 2nd](#)
[_RANK\(\)](#)
[_REGR_AVGX 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)
[_REGR_AVGY 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[_REGR_COUNT 2nd 3rd 4th 5th 6th 7th 8th](#)
[_REGR_INTERCEPT 2nd 3rd 4th 5th 6th 7th](#)
[_REGR_R2 2nd 3rd 4th 5th 6th](#)
[_REGR_SLOPE 2nd 3rd 4th 5th](#)
[_REGR_SXX 2nd 3rd 4th](#)
[_REGR_SXY 2nd 3rd](#)
[_REGR_SYY 2nd](#)
[_REPLACE](#)
[_ROLE_CONTAIN](#)
[_ROLE_ID](#)
[_ROLE_NAME](#)
[_ROW_NUMBER](#)
[_ROWCNT](#)
[_ROWCOUNT_BIG\(.\)](#)
[_scalar 2nd](#)
[_built-in](#)
[_categories of](#)
[_SESSION_USER](#)
[_SHOW_ROLE\(.\)](#)
[_SHOW_SEC_SERVICES\(.\)](#)
[_SORTKEY](#)
[_SQL Server supported \[See SQL Server, functions\]](#)
[_SQL2003_numeric](#)
[_SQRT](#)
[_STATS_DATE](#)
[_STDDEV_POP 2nd](#)
[_STDDEV_SAMP 2nd](#)
[_STDEV](#)
[_STDEVP](#)
[_string](#)
[_SUBSTRING](#)
[_SUM 2nd](#)
[_SUSER_ID](#)
[_SUSER_NAME](#)
[_SUSER_SID](#)
[_SUSER_SNAME](#)
[_SYB_SENDSMSG](#)
[_Sybase adaptive server versus SQL Server](#)
[_Sybase adaptive server-supported](#)
[_SYSTEM_USER](#)
[_TO_UNICHAR](#)
[_TRANSLATE](#)
[_TRIM](#)
[_TSEQUEL](#)
[_TYPEPROPERTY](#)
[_types of](#)
[_UHIGHSURR](#)
[_UOWSURR](#)
[_UNICODE](#)
[_UPPER](#)
[_USCALAR](#)
[_VALID_NAME](#)
[_VALID_USER](#)
[_VAR](#)
[_VAR_POP 2nd](#)
[_VAR_SAMP 2nd](#)
[_VARP](#)
[_WIDTH_BUCKET](#)
[_window 2nd](#)

windowing

YEAR



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[GENERATE_UNIQUE\(\) function \(DB2\)](#)

[GET DIAGNOSTIC statement](#)

[Get methods \(ADO.NET DataReader\)](#)

[Get methods \(JDBC ResultSet\)](#)

[GET_LOCK function \(MySQL\)](#)

[GETANSINULL function](#)

[_SQL Server](#)

[GETDATE\(\) function \(SQL Server\)](#)

[GETHINT function \(DB2\)](#)

[GETUTCDATE\(\) function](#)

[_SQL Server](#)

[GRANT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[GRAPHIC datatype](#)

[_DB2](#)

[_platform comparison](#)

[GRAPHIC function \(DB2\)](#)

[greater than \(>\) operator](#)

[greater than or equal to \(>=\) operator](#)

[GREATEST function](#)

[_MySQL](#)

[_Oracle](#)

[GROUP BY clause 2nd](#)

[group clause](#)

[GROUP_ID\(\) function \(Oracle\)](#)

[GROUPING function](#)

[_Oracle](#)

[_SQL Server](#)

[GROUPING_ID function \(Oracle\)](#)



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[HASHEDVALUE function \(DB2\)](#)

[HAVING clause](#)

[HEIGHT function \(PostgreSQL\)](#)

HEX function

[_DB2](#)

[_MySQL](#)

[HEXTOINT function](#)

[HEXTORAW function \(Oracle\)](#)

[HOST function \(PostgreSQL\)](#)

[HOST_ID\(.\) function \(SQL Server\)](#)

[HOST_NAME\(.\) function \(SQL Server\)](#)

HOUR function

[_DB2](#)

[_MySQL](#)



< Day Day Up >



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[IDENT_CURRENT function](#)

[_SQL Server](#)

[IDENT_INCR function](#)

[_SQL Server](#)

[IDENT_SEED function](#)

[_SQL Server](#)

[identifier rules](#)

[identifiers](#)

[_allowed_characters](#)

[_beginning_characters](#)

[_quoted](#)

[_reserved](#)

[_size](#)

[IDENTITY function](#)

[_SQL Server](#)

[IDENTITY_VAL_LOCAL\(\) function \(DB2\)](#)

[IF function \(MySQL\)](#)

[IFNULL function \(MySQL\)](#)

[IMAGE datatype](#)

[_platform_comparison](#)

[_SQL Server](#)

[implicit transactions](#)

[IN operator 2nd](#)

[_keywords](#)

[_platform_support_for](#)

[_quick_reference](#)

[_SQL2003_syntax](#)

[INDEX_COL function \(SQL Server\)](#)

[INDEXPROPERTY function](#)

[_SQL Server](#)

[INET datatype](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[INIT.ORA file](#)

[INITCAP function](#)

[_Oracle](#)

[INITCAP function \(PostgreSQL\)](#)

[INNER JOIN](#)

[INSERT function](#)

[_DB2](#)

[_MySQL](#)

[INSERT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_operators_and_2nd](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003_syntax](#)

[INSTR function](#)

[_MySQL](#)

[_Oracle](#)

[INT datatype](#)

[_DB2](#)

[_MySQL](#)

[_platform comparison](#)

[_SQL Server](#)

[Int parameter object type \(ADO.NET\)](#)

[INT2 datatype](#)

[_platform comparison](#)

[INT4 datatype](#)

[_platform comparison](#)

[INT8 datatype](#)

[_PostgreSQL](#)

[INTEGER datatype](#)

[_DB2](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[INTEGER function](#)

[_DB2](#)

[_PostgreSQL](#)

[INTERSECT set operator](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[INTERVAL datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[INTERVAL DAY datatype](#)

[_Oracle](#)

[INTERVAL DAY TO SECOND datatype](#)

[_platform comparison](#)

[INTERVAL function](#)

[_MySQL](#)

[_PostgreSQL](#)

[INTERVAL YEAR datatype](#)

[_Oracle](#)

[INTERVAL YEAR TO MONTH datatype](#)

[_platform comparison](#)

[INTTOHEX function](#)

[IS operator](#)

[_keywords](#)

[_platform support for](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL2003 syntax](#)

[IS_FREE_LOCK function \(MySQL\)](#)

[IS_MEMBER function](#)

[_SQL_Server](#)
[IS_SEC_SERVICE_ON function](#)
[IS_SRVROLEMEMBER function](#)
[_SQL_Server](#)
[ISCLOSED function \(PostgreSQL\)](#)
[ISDATE function](#)
[_SQL_Server](#)
[ISFINITE function \(PostgreSQL\)](#)
[ISNULL function](#)
[_MySQL](#)
[_SQL_Server](#)
[ISNUMERIC function](#)
[_SQL_Server](#)
[ISOPEN function \(PostgreSQL\)](#)
[ITERATION_NUMBER function \(Oracle\)](#)





[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[Java Virtual Machine \(JVM\) class loader](#)

JDBC

- [_beginning_transactions](#)
- [_bound_parameters](#)
- [_closing_connection](#)
- [_committing_transactions](#)
- [_database_connection](#)
- [_database_program_example](#)
- [_error_handling](#)
- [_executing_statements](#)
- [_PreparedStatement_set_methods](#)
- [_ResultSet_methods](#)
- [_Get](#)
- [_retrieving_data](#)
- [_rolling_back_transactions](#)

[JDBC \(Java Database Connectivity\)](#)

[JOIN clause 2nd](#)

[JOIN subclause 2nd](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform_support_for](#)
- [_PostgreSQL](#)
- [_programming_tips](#)
- [_rules](#)
- [_SQL_Server](#)
- [_SQL2003_syntax](#)
- [_types](#)

[JULIAN_DAY function \(DB2\)](#)





< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[keywords 2nd](#)

[_DB2 2nd](#)

[_Microsoft SQL Server](#)

[_MySQL 2nd](#)

[_Oracle 2nd](#)

[_PostgreSQL 2nd](#)

[_shared 2nd](#)

[_SQL Server](#)

[_SQL2003 2nd](#)

[_Sybase adaptive server 2nd 3rd 4th 5th](#)



< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[LAG function \(Oracle\)](#)

[LAST function \(Oracle\)](#)

[LAST_DAY function \(Oracle\)](#)

[LAST_INSERT_ID function \(MySQL\)](#)

[LAST_VALUE function \(Oracle\)](#)

[LCASE function \(MySQL\)](#)

[lct_admin function](#)

[LDAP \(Lightweight Directory Access Protocol\)](#)

[_on DB2 servers](#)

[LEAD function \(Oracle\)](#)

[LEAST function](#)

[_MySQL](#)

[_Oracle](#)

[LEFT \[OUTER\] JOIN](#)

[LEFT function](#)

[_DB2](#)

[_MySQL](#)

[_SQL Server](#)

[LEN function](#)

[_SQL Server](#)

[LENGTH function](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[LENGTHB function \(Oracle\)](#)

[less than \(<\) operator](#)

[less than or equal to \(<=\) operator](#)

[LICENSE_ENABLED function](#)

[LIKE operator 2nd](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[LINE datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[literals 2nd](#)

[LN function](#)

[LNNVL function \(Oracle\)](#)

[LOAD_FILE function \(MySQL\)](#)

[LOCALTIMESTAMP function \(Oracle\)](#)

[LOCATE function](#)

[_DB2](#)

[_MySQL](#)

[LOCATE\(substr,str,pos\)](#)

[LOG function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL_Server](#)

LOG10 function

- [_DB2](#)
- [_MySQL](#)
- [_SQL_Server](#)

[LOG2 function \(MySQL\)](#)

[logfiles \(Oracle\)](#)

[logical operators 2nd](#)

LONG datatype

- [_Oracle](#)
- [_platform comparison](#)

LONG RAW datatype

- [_Oracle](#)
- [_platform comparison](#)

LONG VARCHAR datatype

- [_DB2](#)
- [_platform comparison](#)

LONG VARCHAR FOR BIT DATA datatype

- [_DB2](#)

LONG VARGRAPHIC datatype

- [_DB2](#)
- [_platform comparison](#)

[LONG_VARCHAR function \(DB2\)](#)

[LONG_VARGRAPHIC function \(DB2\)](#)

LONGBLOB datatype

- [_MySQL](#)
- [_platform comparison](#)

LONGTEXT datatype

- [_MySQL](#)
- [_platform comparison](#)

[LOWER function](#)

LPAD function

- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)

LSEG datatype

- [_platform comparison](#)
- [_PostgreSQL](#)

[LSEG function \(PostgreSQL\)](#)

LTRIM function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL_Server](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

MACADDR datatype

[_platform_comparison](#)

[_PostgreSQL](#)

[MAKE_REF function \(Oracle\)](#)

[MAKE_SET function \(MySQL\)](#)

[MASKLEN function \(PostgreSQL\)](#)

[MAX function 2nd](#)

[MD5 function \(MySQL\)](#)

[MEDIAN function \(Oracle\)](#)

MEDIUMBLOB datatype

[_MySQL](#)

[_platform_comparison](#)

MEDIUMINT datatype

[_MySQL](#)

MEDIUMTEXT datatype

[_MySQL](#)

[_platform_comparison](#)

[MERGE statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL_Server](#)

[_SQL2003_syntax](#)

[metadata](#)

[methods](#)

[_ADO.NET_DataReader](#)

[_JDBC_ResultSet](#)

[MICROSECOND function \(DB2\)](#)

[Microsoft SQL Server](#)

[Microsoft SQL Server keywords](#)

[MID function \(MySQL\)](#)

[MIDNIGHT_SECONDS function \(DB2\)](#)

[MIN function 2nd](#)

MINUTE function

[_DB2](#)

[_MySQL](#)

[MOD function](#)

[modula \(%\) arithmetic operator \(SQLServer\)](#)

MONEY datatype

[_platform_comparison](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[MONTH function](#)

[_DB2](#)

[_MySQL](#)

[_SQL_Server](#)

MONTHNAME function

[_DB2](#)

MySQL

MONTHS_BETWEEN function (Oracle)

multiplication (*) arithmetic operator

MULTIPLY_ALT function (DB2)

MUT_EXCL_ROLES function

MySQL

_allowed characters in identifiers

ALTER TABLE syntax

_beginning identifier character

CALL statement

COMMIT statement

_connecting using JDBC

CREATE/ALTER DATABASE statements

CREATE/ALTER FUNCTION/PROCEDURE statements

CREATE/ALTER INDEX statements

CREATE/ALTER METHOD statements

CREATE/ALTER ROLE statements

CREATE/ALTER SCHEMA statements

CREATE/ALTER TABLE

_keywords and parameters

CREATE/ALTER TABLE statements

ALTER

ALTER COLUMN

AUTO_INCREMENT

AVG_ROW_LENGTH

CHANGE

CHECKSUM

COMMENT

_constraint_type

DATA DIRECTORY

DELAY_KEY_WRITE

DROP

ENABLE KEYS

IF NOT EXISTS

IGNORE SELECT_statement

INDEX DIRECTORY

INSERT_METHOD

_keywords

MAX_ROWS

MIN_ROWS

MODIFY

PACK_KEYS

_parameters

PASSWORD

RAID_CHUNKS

RAID_CHUNKSIZE

RAID_TYPE

RENAME

REPLACE SELECT_statement

ROW_FORMAT

TEMPORARY

_TYPE

CREATE/ALTER TRIGGER statements

CREATE/ALTER TYPE statements

CREATE/ALTER VIEW statements

_datatypes

DECLARE CURSOR command

DELETE statement

DISCONNECT statement

DROP statements
EXCEPT set operator
FETCH statement
functions
_,POW
_ACOS
_ADDDATE
_ASCII
_ASIN
_ATAN
_ATAN2
_BENCHMARK
_BIN
_BINARY
_BIT_AND
_BIT_COUNT
_BIT_OR
_CHAR
_COALESCE
_COMPRESS
_CONCAT_WS
_CONNECTION_ID(.)
_CONV
_COS
_COT
_CURDATE(.)
_CURTIME(.)
_DATABASE(.)
_DATE_FORMAT
_DATE_ADD
_DATE_SUB
_DAYNAME
_DAYOFMONTH
_DAYOFWEEK
_DAYOFYEAR
_DECODE
_DEGREES
_ELT
_ENCODE
_ENCRYPT
_EXPORT_SET
_FIELD
_FIND_IN_SET
_FORMAT
_FROM_DAYS
_FROM_UNIXTIME 2nd
_GET_LOCK
_GREATEST
_HEX
_HOUR
_IF
_IFNULL
_INSERT
_INSTR
_INTERVAL
_IS_FREE_LOCK
_ISNULL
_LAST_INSERT_ID
_LCASE

LEAST
LEFT
LENGTH
LOAD_FILE
LOCATE
LOG
LOG10
LOG2
LPAD
LTRIM
MAKE_SET
MD5
MID
MINUTE
MONTH
MONTHNAME
NOW()
NULLIF
OCT
ORD
PASSWORD
PERIOD_ADD
PERIOD_DIFF
PI()
POSITION
POWER
QUARTER
RADIANS
RAND()
RELEASE_LOCK
REPEAT
REPLACE
REVERSE
RIGHT
ROUND
RPAD
RTRIM
SEC_TO_TIME
SECOND
SESSION_USER()
SHA
SHA1
SIGN
SIN
SOUNDEX
SPACE
STD
STDDEV
STRCMP
SUBDATE
SUBSTRING
SUBSTRING_INDEX
SYSDATE()
SYSTEM_USER()
TAN
TIME_FORMAT
TIME_TO_SEC
TO_DAYS
TRUNCATE

- [_UCASE](#)
- [_UNCOMPRESS](#)
- [_UNHEX](#)
- [_UNIX_TIMESTAMP](#)
- [_USER\(\)](#)
- [_VERSION\(\)](#)
- [_WEEK](#)
- [_WEEKDAY](#)
- [_YEAR](#)
- [_YEARWEEK](#)
- [_GRANT statement](#)
- [_identifier size](#)
- [_INSERT statement](#)
- [_INTERSECT set operator](#)
- [_JOIN subclause](#)
- [_keywords](#)
- [_LIKE operator](#)
- [_MERGE statement](#)
- [_OPEN statement](#)
- [_ORDER BY clause](#)
- [_RELEASE SAVEPOINT statement](#)
- [_reserved identifiers](#)
- [_RETURN statement](#)
- [_REVOKE statement](#)
- [_ROLLBACK statement](#)
- [_SAVEPOINT statement](#)
- [_schema addressing](#)
- [_SELECT statement](#)
 - [_{FOR UPDATE | LOCK IN SHARE MODE}](#)
 - [_FROM É USE INDEX](#)
 - [_HIGH_PRIORITY](#)
 - [_IGNORE INDEX](#)
 - [_INTO](#)
 - [_JOIN syntax](#)
 - [_LIMIT](#)
 - [_PROCEDURE](#)
 - [_select_item](#)
 - [_SQL_BIG_RESULT](#)
 - [_SQL_BUFFER_RESULT](#)
 - [_SQL_SMALL_RESULT](#)
 - [_STRAIGHT_JOIN](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_special rules](#)
- [_START TRANSACTION statement](#)
- [_SUBQUERY substatement](#)
- [_TRUNCATE TABLE statement](#)
- [_UNION set operator](#)
- [_UPDATE statement](#)
- [_web site](#)
- [_WHERE clause](#)



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[naming conventions](#)

[NANVL function \(Oracle\)](#)

NATIONAL CHAR datatype

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL CHAR VARYING datatype

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL CHARACTER datatype

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL CHARACTER VARYING datatype

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

NATIONAL TEXT datatype

[_platform comparison](#)

[_SQL Server](#)

[NATURAL \[INNER | {LEFT | RIGHT}\[OUTER\] \] JOIN](#)

NCHAR datatype

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_SQL Server](#)

[NCHAR function](#)

[_Oracle](#)

[_SQL Server](#)

NCHAR VARYING datatype

[_Oracle](#)

[_platform comparison](#)

NCHAR() datatype

[_SQL Server 2nd](#)

[_Sybase adaptive server](#)

NCLOB datatype

[_Oracle](#)

[_platform comparison](#)

[nested parentheses and operator precedence](#)

nested tables

[_in FROM clause](#)

[_Oracle](#)

[_subqueries](#)

[NETMASK function \(PostgreSQL\)](#)

[NETWORK function \(PostgreSQL\)](#)

[NEW_TIME function \(Oracle\)](#)

[NEWID\(\) function](#)

[_SQL Server](#)

[NEXT_DAY function \(Oracle\)](#)

[NLS_CHARSET_DECL_LEN function \(Oracle\)](#)

[NLS_CHARSET_ID function \(Oracle\)](#)

[NLS_CHARSET_NAME function \(Oracle\)](#)

- [NLS_INITCAP function \(Oracle\)](#)
- [NLS_LOWER function \(Oracle\)](#)
- [NLS_UPPER function \(Oracle\)](#)
- [NLSSORT function \(Oracle\)](#)
- [nondeterministic functions](#)
- [not equal to \(!=\) operator 2nd 3rd](#)
- [not greater than \(!>\) operator](#)
- [not less than \(!<\) operator](#)
- [NOT operator](#)
- [NOW\(\) function \(MySQL\)](#)
- [NPOINTS function \(PostgreSQL\)](#)
- [NTEXT datatype](#)
 - [_platform comparison](#)
 - [_SQL Server 2nd](#)
- [NTILE function \(Oracle\)](#)
- [NULL values](#)
- [NULLIF function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
 - [_SQL Server](#)
- [NUM datatype](#)
 - [_DB2](#)
- [NUMBER datatype](#)
 - [_Oracle](#)
 - [_platform comparison](#)
- [NUMERIC datatype](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform comparison](#)
 - [_PostgreSQL 2nd](#)
 - [_SQL Server 2nd](#)
- [numeric datatype and unary operators](#)
- [numeric functions](#)
 - [_enhanced](#)
- [NUMTODSINTERVAL function \(Oracle\)](#)
- [NUMTOYMINTERVAL function \(Oracle\)](#)
- [NVARCHAR\(\) datatype \(Sybase Adaptive Server\)](#)
- [NVARCHAR datatype](#)
 - [_MySQL](#)
 - [_platform comparison](#)
 - [_SQL Server](#)
- [NVARCHAR2 datatype](#)
 - [_Oracle](#)
- [NVARCHAR2\(\) datatype](#)
 - [_platform comparison](#)
- [NVL function \(Oracle\)](#)
- [NVL2 function \(Oracle\)](#)

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[OBJECT_ID function \(SQL Server\)](#)

[OBJECT_NAME function \(SQL Server\)](#)

[OBJECTPROPERTY function](#)

[_SQL Server](#)

[objects](#)

[OCT function \(MySQL\)](#)

[OCTET_LENGTH function](#)

[OdbcConnection](#)

[_connection_string_attributes_for](#)

[OID datatype](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[OLAP functions](#)

[OleDbConnection](#)

[_connection_string_attributes_for](#)

[OPEN function](#)

[_SQL Server](#)

[OPEN statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform_support_for](#)

[_PostgreSQL](#)

[_programming_tips](#)

[_quick_reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003_syntax](#)

[OPENDATASOURCE function](#)

[_SQL Server](#)

[OPENQUERY function](#)

[_SQL Server](#)

[OPENROWSET function](#)

[_SQL Server](#)

[operator precedence](#)

[operators 2nd](#)

[_string](#)

[OPERATORS, quick reference](#)

[OR operator](#)

[ORA_HASH function \(Oracle\)](#)

[Oracle](#)

[_allowed_characters_in_identifiers](#)

[_ALTER INDEX syntax](#)

[_ALTER TABLE statement](#)

[_ALTER TABLE syntax](#)

[_altering_a_partitioned_table](#)

[_basics](#)

[_beginning_identifier_character](#)

[_CALL statement](#)

[_column_constraint_clause](#)

[_COMMIT statement](#)

[_compressed_tables](#)

[_CONNECT statement](#)

[_connecting_using JDBC](#)
[_CREATE_INDEX syntax](#)
[_CREATE/ALTER DATABASE statements](#)
[_ACTIVATE](#)
[_ADD LOGFILE](#)
[_ADD SUPPLEMENTAL LOG DATA](#)
[_ALLOW](#)
[_ARCHIVELOG 2nd](#)
[_AUTOMATIC](#)
[_BACKUP CONTROLFILE TO](#)
[_BEGIN BACKUP](#)
[_BIGFILE](#)
[_CANCEL](#)
[_CHARACTER SET](#)
[_CLEAR LOGFILE](#)
[_CONTINUE](#)
[_CONTROLFILE REUSE](#)
[_CREATE DATAFILE](#)
[_CREATE STANDBY CONTROLFILE AS](#)
[_DATABASE](#)
[_DATAFILE 2nd](#)
[_DEFAULT TABLESPACE](#)
[_DEFAULT TEMPORARY TABLESPACE](#)
[_ENABLE BLOCK CHANGE TRACKING](#)
[_ENABLE THREAD](#)
[_EXTENT MANAGEMENT 2nd](#)
[_FLASHBACK](#)
[_FOR](#)
[_FORCE LOGGING 2nd](#)
[_GUARD](#)
[_LOGFILE 2nd](#)
[_MANAGED STANDBY DATABASE](#)
[_MAXDATAFILES](#)
[_MAXINSTANCES](#)
[_MAXLOGFILES](#)
[_MAXLOGHISTORY](#)
[_MAXLOGMEMBERS](#)
[_MOUNT](#)
[_NATIONAL CHARACTER SET](#)
[_OPEN](#)
[_PARALLEL](#)
[_RECOVER](#)
[_REGISTER](#)
[_RENAME FILE](#)
[_RENAME GLOBAL_NAME TO](#)
[_SET DEFAULT TABLESPACE](#)
[_SET STANDBY](#)
[_SET TIME_ZONE](#)
[_SMALLFILE](#)
[_STANDBY](#)
[_START LOGICAL STANDBY APPLY](#)
[_SYSAUX DATAFILE](#)
[_TABLESPACE](#)
[_TEST](#)
[_UNDO TABLESPACE](#)
[_UNTIL](#)
[_USER SYS IDENTIFIED BY](#)
[_USER SYSTEM IDENTIFIED BY](#)
[_CREATE/ALTER FUNCTION/PROCEDURE statements](#)

AGGREGATE
ALTER
AS
AUTHID
code_block
COMPILE
CREATE
IN
IS
LANGUAGE
NOCOPY
OUT
PARALLEL_ENABLE
PIPELINED
CREATE/ALTER INDEX statements
ALLOCATE EXTENT
alter_index_partitioning_clause
ASC
BITMAP
CLUSTER
COALESCE
COMPRESS
COMPUTE
DEALLOCATE UNUSED
DESC
ENABLE
GLOBAL
INDEXTYPE IS
LOCAL
LOGGING
MONITORING USAGE
NOSORT
PARALLEL
PARAMETERS
physical_attributes_clause
rebuild_clause
RENAME TO
REVERSE
SHRINK SPACE
TABLESPACE DEFAULT
UNUSABLE
UPDATE BLOCK REFERENCES
CREATE/ALTER METHOD statements
CREATE/ALTER ROLE statements
CREATE/ALTER SCHEMA statements
CREATE/ALTER TABLE statements
ADD É
ADD PARTITION
ALLOCATE EXTENT
AS object_type
AS subquery
CACHE
CLUSTER
COALESCE
COALESCE PARTITION
COMPRESS
constraint_state clause
DEALLOCATE UNUSED
DROP É

DROP PARTITION
ENABLE ROW MOVEMENT
ENABLE TABLE LOCK
EXCHANGE PARTITION
foreign_key_constraint
GROUP
inline_ref_constraint_2nd
LOGGING
MERGE PARTITION
MINIMIZE RECORDS_PER_BLOCK
MODIFY DEFAULT ATTRIBUTES
MODIFY É
MODIFY PARTITION
MODIFY SUBPARTITION
MONITORING
MOVE É
MOVE PARTITION
OBJECT IDENTIFIER IS
OF XMLTYPE
OIDINDEX
ON COMMIT
ORGANIZATION EXTERNAL
ORGANIZATION HEAP
ORGANIZATION INDEX
PARALLEL
parameters
partition_clause
partition_level_subpartition_clause
partition_storage_clause
physical_attributes_clause
PROJECT COLUMN
RENAME É
RENAME PARTITION
ROWDEPENDENCIES
SET SUBPARTITION TEMPLATE
SET UNUSED É
SHRINK SPACE
SPLIT PARTITION
storage_clause_2nd
SUBSTITUTABLE AT ALL LEVELS
table_constraint_clause
table_partition_description_clause
table_ref_constraint_2nd
TABLESPACE
TRUNCATE PARTITION
update_index_clause
UPGRADE INCLUDING DATA
xml_schema_spec
XMLTYPE
CREATE/ALTER TRIGGER statements
CREATE/ALTER TYPE statements
ADD
AS OBJECT
AS TABLE OF
AS VARRAY
AUTHID
BODY
CASCADE
COMPILE

[_compiler_directives_2nd](#)
[_CONVERT_TO_SUBSTITUTABLE](#)
[_datatype](#)
[_DEBUG](#)
[_DROP](#)
[_EXCEPTIONS INTO](#)
[_EXTERNAL_NAME](#)
[_FORCE](#)
[_INCLUDING TABLE DATA](#)
[_INVALIDATE](#)
[_MEMBER](#)
[_MODIFY](#)
[_MODIFY_ELEMENT_TYPE](#)
[_MODIFY_LIMIT](#)
[_OID](#)
[_OR_REPLACE](#)
[_OVERRIDING](#)
[_REPLACE AS OBJECT](#)
[_REUSE_SETTINGS](#)
[_SPECIFICATION](#)
[_STATIC](#)
[_UNDER](#)
[_CREATE/ALTER VIEW statements](#)
[_datatypes](#)
[_DECLARE_CURSOR_command](#)
[_default_database_user_accounts](#)
[_DELETE_statement](#)
[_DISCONNECT_statement](#)
[_DROP statements](#)
[_\[PUBLIC\] DATABASE LINK](#)
[_\[PUBLIC\] SYNONYM](#)
[_CLUSTER](#)
[_CONTEXT](#)
[_DATABASE](#)
[_DIMENSION](#)
[_DIRECTORY](#)
[_FUNCTION](#)
[_INDEX](#)
[_INDEXTYPE](#)
[_JAVA](#)
[_LIBRARY](#)
[_MATERIALIZED_VIEW](#)
[_OPERATOR](#)
[_OUTLINE](#)
[_PACKAGE](#)
[_PROCEDURE](#)
[_PROFILE](#)
[_ROLE](#)
[_ROLLBACK_SEGMENT](#)
[_SEQUENCE](#)
[_SNAPSHOT](#)
[_TABLE](#)
[_TABLESPACE](#)
[_TRIGGER](#)
[_TYPE](#)
[_USER](#)
[_VIEW](#)
[_EXCEPT set operator](#)
[_FETCH_statement](#)

functions

ACOS
ADD_MONTHS
ASCII
ASCIISTR
ASIN
ATAN
ATAN2
BFILENAME
BIN_TO_NUM
BITAND
CARDINALITY
CHARTOROWID
CHR
COALESCE
COLLECT
COMPOSE
CONCAT
CONVERT
CORR_K
COS
COSH
CV
DBTIMEZONE
DECODE
DECOMPOSE
DEPTH
DEREF
DUMP
EMPTY_BLOB()
EMPTY_CLOB()
EXISTSNODE
EXTRACT
EXTRACTVALUE
FIRST
FIRST_VALUE
FROM_TZ
GREATEST
GROUP_ID()
GROUPING
GROUPING_ID
HEXTORAW
INITCAP
INSTR
ITERATION_NUMBER
LAG
LAST
LAST_DAY
LAST_VALUE
LEAD
LEAST
LENGTH
LENGTHB
LNNVL
LOCALTIMESTAMP
LOG
LPAD
LTRIM
MAKE_REF

MEDIAN
MONTHS_BETWEEN
NANVL
NCHAR
NEW_TIME
NEXT_DAY
NLS_CHARSET_DECL_LEN
NLS_CHARSET_ID
NLS_CHARSET_NAME
NLS_INITCAP
NLS_LOWER
NLS_UPPER
NLSSORT
NTILE
NULLIF
NUMTODSINTERVAL
NUMTOYMINTERVAL
NVL
NVL2
ORA_HASH
PATH
POWERMULTISET
PRESENTNNV
PRESENTV
PREVIOUS
RATIO_TO_REPORT
RAWTOHEX
REF
REFTOHEX
REGEXP_INSTR
REGEXP_REPLACE
REGEXP_SUBSTR
REMAINDER
REPLACE
ROUND
ROWIDTOCHAR
RPAD
RTRIM
SCN_TO_TIMESTAMP
SESSIONTIMEZONE
SET
SIGN
SIN
SINH
SOUNDEX
STATS_BINOMIAL_TEST
STDDEV
STDDEV_SAMP
STDEV_POP
SUBSTR
SUBSTRB
SYS_CONNECT_BY_PATH
SYS_CONTEXT
SYS_DBURIGEN
SYS_EXTRACT_UTC
SYS_GUID()
SYS_TYPEID
SYS_XMLAGG
SYS_XMLGEN

_SYSDATE
_SYSTIMESTAMP
_TAN
_TANH
_TIMESTAMP_TO_SCN
_TO_BINARY_DOUBLE
_TO_BINARY_FLOAT
_TO_CHAR 2nd
_TO_CLOB
_TO_DATE
_TO_DSINTERVAL
_TO_LOB
_TO_MULTI_BYTE
_TO_NCHAR
_TO_NCLOB
_TO_NUMBER
_TO_SINGLE_BYTE
_TO_TIMESTAMP
_TO_TIMESTAMP_TZ
_TO_YMINTERVAL
_TRANSLATE 2nd
_TREAT
_TRUNC 2nd
_TZ_OFFSET
_UID
_UNISTR
_UPDATEXML
_USERENV
_VALUE
_VAR_POP
_VAR_SAMP
_VARIANCE
_VSIZE
_XMLAGG
_XMLCOLATTVAL
_XMLCONCAT
_XMLELEMENT
_XMLFOREST
_XMLSEQUENCE
_XMLTRANSFORM
._global temporary table
._GRANT statement
 _IDENTIFIED BY
 _object_privilege
 _ON
 _role
 _system_privilege
 _TO
._identifier size
._index-organized table (IOT)
._indexes
._INIT.ORA file
._INSERT statement
 _ALL
 _conditional multitable
 _ELSE
 _FIRST
 _INSERT
 _subquery

- [_TABLE](#)
- [_table_name](#)
- [_unconditional_multitable](#)
- [_VALUES](#)
- [_WHEN](#)
- [_INTERSECT_set_operator](#)
- [_JOIN_subclause](#)
- [_keywords](#)
- [_LIKE_operator](#)
- [_local_index_partitioning](#)
- [_logfiles](#)
- [_MERGE_statement](#)
- [_nested_tables](#)
- [_object-type_tables](#)
- [_OPEN_statement](#)
- [_ORDER_BY_clause](#)
- [_ORGANIZATION_HEAP_clause](#)
- [_partitioned_and_subpartitioned_tables](#)
 - [_altering](#)
- [_physical_attributes_clause](#)
- [_primary_databases](#)
- [_RELEASE_SAVEPOINT_statement](#)
- [_reserved_identifiers](#)
- [_RETURN_statement](#)
- [_REVOKE_statement](#)
- [_ROLLBACK_statement](#)
- [_SAVEPOINT_statement](#)
- [_schema_addressing](#)
- [_SELECT_statement](#)
 - [_\[CROSS\]_JOIN](#)
 - [_ALL](#)
 - [_AS](#)
 - [_DISTINCT](#)
 - [_flashback_queries](#)
 - [_flashback_query_clause](#)
 - [_FOR_UPDATE](#)
 - [_FROM](#)
 - [_FULL_\[OUTER\]_JOIN](#)
 - [_INNER_JOIN](#)
 - [_INTO](#)
 - [_JOIN](#)
 - [_LEFT_\[OUTER\]_JOIN](#)
 - [_MODEL_clause](#)
 - [_NATURAL](#)
 - [_ON](#)
 - [_optimizer_hint](#)
 - [_ORDER_SIBLINGS_BY](#)
 - [_PARTITION](#)
 - [_PARTITION_BY](#)
 - [_partitioned_outer_joins](#)
 - [_RIGHT_\[OUTER\]_JOIN](#)
 - [_SAMPLE](#)
 - [_select_item](#)
 - [_SUBPARTITION](#)
 - [_subquery](#)
 - [_TABLE](#)
 - [_UNIQUE](#)
 - [_USING](#)
 - [_VERSIONS_BETWEEN](#)

- [_WHERE](#)
- [_WITH query_name AS](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME_ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_special rules](#)
- [_standby databases](#)
- [_START TRANSACTION statement](#)
- [_storage_clause and LOBs](#)
- [_stored procedures](#)
- [_SUBQUERY substatement](#)
- [_syntax for object table](#)
- [_syntax for XMLType table](#)
- [_syntax to alter existing database](#)
- [_syntax to create new database](#)
- [_system privileges](#)
 - [_{ALTER | DROP} ANY ROLE](#)
 - [_{CREATE | ALTER | DROP | EXECUTE} ANY PROCEDURE](#)
 - [_{CREATE | ALTER | DROP | SELECT} ANY SEQUENCE](#)
 - [_{CREATE | ALTER | DROP} ANY CLUSTER](#)
 - [_{CREATE | ALTER | DROP} ANY DIMENSION](#)
 - [_{CREATE | ALTER | DROP} ANY INDEX](#)
 - [_{CREATE | ALTER | DROP} ANY MATERIALIZED VIEW](#)
 - [_{CREATE | ALTER | DROP} ANY SNAPSHOT](#)
 - [_{CREATE | ALTER | DROP} ANY TYPE](#)
 - [_{CREATE | ALTER | DROP} PROFILE](#)
 - [_{CREATE | ALTER | DROP} ROLLBACK SEGMENT](#)
 - [_{CREATE | ALTER | DROP} TABLESPACE](#)
 - [_{CREATE | DROP | ALTER} ANY OUTLINE](#)
 - [_{CREATE | DROP | EXECUTE} ANY INDEXTYPE](#)
 - [_{CREATE | DROP | EXECUTE} ANY OPERATOR](#)
 - [_{CREATE | DROP} ANY DIRECTORY](#)
 - [_{CREATE | DROP} ANY CONTEXT](#)
 - [_{CREATE | DROP} ANY LIBRARY](#)
 - [_{CREATE | DROP} ANY SYNONYM](#)
 - [_{CREATE | DROP} ANY TRIGGER](#)
 - [_{CREATE | DROP} ANY VIEW](#)
 - [_{CREATE | DROP} PUBLIC SYNONYM](#)
- [_ADMINISTER DATABASE TRIGGER](#)
- [_ALTER ANY TABLE](#)
- [_ALTER ANY TRIGGER](#)
- [_ALTER DATABASE](#)
- [_ALTER RESOURCE COST](#)
- [_ALTER SESSION](#)
- [_ALTER SYSTEM](#)
- [_ALTER USER](#)
- [_ANALYZE ANY](#)
- [_AUDIT ANY](#)
- [_AUDIT SYSTEM](#)
- [_BACKUP ANY TABLE](#)
- [_BECOME USER](#)
- [_COMMENT ANY TABLE](#)
- [_CREATE ANY TABLE](#)

[_CREATE CLUSTER](#)
[_CREATE DATABASE LINK](#)
[_CREATE DIMENSION](#)
[_CREATE INDEXTYPE](#)
[_CREATE MATERIALIZED VIEW](#)
[_CREATE OPERATOR](#)
[_CREATE PROCEDURE](#)
[_CREATE PUBLIC DATABASE LINK](#)
[_CREATE ROLE](#)
[_CREATE SEQUENCE](#)
[_CREATE SESSION](#)
[_CREATE SNAPSHOT](#)
[_CREATE SYNONYM](#)
[_CREATE TRIGGER](#)
[_CREATE TYPE](#)
[_CREATE USER](#)
[_CREATE VIEW](#)
[_DEBUG ANY PROCEDURE](#)
[_DEBUG CONNECT SESSION](#)
[_DELETE ANY TABLE](#)
[_DROP ANY TABLE](#)
[_DROP PUBLIC DATABASE LINK](#)
[_DROP USER](#)
[_EXECUTE ANY TYPE](#)
[_EXEMPT ACCESS POLICY](#)
[_FLASHBACK ANY TABLE](#)
[_FORCE ANY TRANSACTION](#)
[_FORCE TRANSACTION](#)
[_GLOBAL QUERY REWRITE](#)
[_GRANT ANY OBJECT PRIVILEGE](#)
[_GRANT ANY PRIVILEGE](#)
[_GRANT ANY ROLE](#)
[_INSERT ANY TABLE](#)
[_LOCK ANY TABLE](#)
[_MANAGE TABLESPACE](#)
[_ON COMMIT REFRESH](#)
[_QUERY REWRITE](#)
[_RESTRICTED SESSION](#)
[_RESUMABLE](#)
[_SELECT ANY DICTIONARY](#)
[_SELECT ANY OUTLINE](#)
[_SELECT ANY TABLE](#)
[_SYSDBA](#)
[_SYSOPER](#)
[_UNDER ANY TYPE](#)
[_UNDER ANY VIEW](#)
[_UNLIMITED TABLESPACE](#)
[_UPDATE ANY TABLE](#)
[_TRUNCATE TABLE statement](#)
[_UDFs](#)
[_UNION set operator](#)
[_UPDATE statement](#)
[_WHERE clause](#)
[_window syntax](#)
[_XMLType tables](#)
[Oracle keywords](#)
[ORD function \(MySQL\)](#)
[ORDER BY clause 2nd 3rd](#)
[_DB2](#)

[_keywords](#)
[_MySQL](#)
[_Oracle](#)
[_platform support for](#)
[_PostgreSQL](#)
[_programming tips](#)
[_quick reference](#)
[_rules](#)
[_SQL Server](#)
[_SQL2003 syntax](#)
[ordering clause](#)
[ORGANIZATION HEAP clause \(Oracle\)](#)
[OVERLAY function](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[parameter object types \(ADO.NET\)](#)

[parameters](#)

[_binding input](#)

[_bound](#) [See bound parameters]

[parametrized statements](#)

[parentheses](#)

[_nested and operator precedence](#)

[PARSENAME function](#)

[_SQL Server](#)

[partitioned tables](#)

[_Oracle](#)

[_altering](#)

[partitioning clause](#)

[PASSWORD function \(MySQL\)](#)

[PATH datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[PATH function](#)

[_Oracle](#)

[_PostgreSQL](#)

[PATINDEX function](#)

[_SQL Server](#)

[PCLOSE function \(PostgreSQL\)](#)

[PERCENT_RANK function 2nd 3rd](#)

[PERCENTILE_CONT function 2nd](#)

[PERCENTILE_DISC function 2nd](#)

[PERIOD_ADD function \(MySQL\)](#)

[PERIOD_DIFF function \(MySQL\)](#)

[permissions](#)

[PERMISSIONS function](#)

[_SQL Server](#)

[physical_attributes_clause \(Oracle\)](#)

[PI\(\) function](#)

[_MySQL](#)

[_PostgreSQL](#)

[_SQL Server](#)

[PL/pgSQL](#)

[PL/SQL](#)

[platform comparisons](#)

[POINT datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[POINT function \(PostgreSQL\)](#)

[POLYGON datatype](#)

[_platform comparison](#)

[_PostgreSQL](#)

[POLYGON function \(PostgreSQL\)](#)

[POPEN function \(PostgreSQL\)](#)

[POSITION function](#)

[_MySQL](#)

[POSSTR function \(DB2\)](#)

[PostgreSQL](#)

[_allowed characters in identifiers](#)

[_beginning identifier character](#)

- [_CLOSE_CURSOR_statement](#)
- [_COMMIT_statement](#)
- [_CONNECT_statement](#)
- [_connecting_using_JDBC](#)
- [_CREATE/ALTER_DATABASE_statements](#)
- [_CREATE/ALTER_FUNCTION/PROCEDURE_statements](#)
- [_CREATE/ALTER_INDEX_statements](#)
- [_CREATE/ALTER_METHOD_statements](#)
- [_CREATE/ALTER_ROLE_statements](#)
- [_CREATE/ALTER_SCHEMA_statements](#)
- [_CREATE/ALTER_TABLE_statements](#)
- [_CREATE/ALTER_TRIGGER_statements](#)
- [_CREATE/ALTER_TYPE_statements](#)
- [_CREATE/ALTER_VIEW_statements](#)
- [_datatypes](#)
- [_DECLARE_CURSOR_command](#)
- [_DELETE_statement](#)
- [_DISCONNECT_statement](#)
- [_DROP_statements](#)
 - [_\[_PROCEDURAL_\]LANGUAGE](#)
 - [_AGGEGATE](#)
 - [_DATABASE](#)
 - [_FUNCTION](#)
 - [_GROUP](#)
 - [_INDEX](#)
 - [_OPERATOR](#)
 - [_RULE](#)
 - [_SEQUENCE](#)
 - [_TABLE](#)
 - [_TRIGGER](#)
 - [_TYPE](#)
 - [_USER](#)
 - [_VIEW](#)
- [_EXCEPT_set_operator](#)
- [_FETCH_statement](#)
- [_functions](#)
 - [_ABSTIME](#)
 - [_ACOS](#)
 - [_AGE](#)
 - [_AREA](#)
 - [_ASCII](#)
 - [_ASIN](#)
 - [_ATAN](#)
 - [_ATAN2](#)
 - [_BOX](#)
 - [_BROADCAST](#)
 - [_CBRT](#)
 - [_CENTER](#)
 - [_CHAR](#)
 - [_CHAR_LENGTH](#)
 - [_CIRCLE](#)
 - [_COALESCE](#)
 - [_COS](#)
 - [_COT_2nd](#)
 - [_DATE_PART](#)
 - [_DATE_TRUNC](#)
 - [_DEGREES](#)
 - [_DIAMETER](#)
 - [_FLOAT](#)

FLOAT4
HEIGHT
HOST
INITCAP
INTEGER
INTERVAL
ISCLOSED
ISFINITE
ISOPEN
LENGTH
LOG
LPAD
LSEG
LTRIM
MASKLEN
NETMASK
NETWORK
NPOINTS
NULLIF
PATH
PCLOSE
PI()
POINT
POLYGON
POPEN
POW
RADIANS
RADIUS
RETIME
ROUND
RPAD
RTRIM
SET_MASKLEN
SIN
SUBSTR
SUBSTRING
TAN
TEXT
TIMESTAMP
TO_CHAR
TO_DATE
TO_NUMBER
TO_TIMESTAMP
TRANSLATE
TRUNC
VARCHAR
WIDTH
GRANT statement
identifier size
INSERT statement
INTERSECT set operator
JOIN subclause
keywords
LIKE operator
MERGE statement
OPEN statement
ORDER BY clause
RELEASE SAVEPOINT statement
reserved identifiers

- [_RETURN_statement](#)
- [_REVOKE_statement](#)
- [_ROLLBACK_statement](#)
- [_SAVEPOINT_statement](#)
- [_schema_addressing](#)
- [_SELECT_statement](#)
 - [_\[CROSS\] JOIN](#)
 - [_ALL](#)
 - [_AS](#)
 - [_FOR_UPDATE_OF](#)
 - [_FROM](#)
 - [_FULL \[OUTER\] JOIN](#)
 - [_GROUP_BY](#)
 - [_INNER JOIN](#)
 - [_INTO](#)
 - [_LEFT \[OUTER\] JOIN](#)
 - [_NATURAL](#)
 - [_ON](#)
 - [_ORDER_BY](#)
 - [_RIGHT \[OUTER\] JOIN](#)
 - [_select_item](#)
- [_SET_CONNECTION_statement](#)
- [_SET_CONSTRAINT_statement](#)
- [_SET_PATH_statement](#)
- [_SET_ROLE_statement](#)
- [_SET_SCHEMA_statement](#)
- [_SET_SESSION_AUTHORIZATION_statement](#)
- [_SET_statement](#)
- [_SET_TIME_ZONE_statement](#)
- [_SET_TRANSACTION_statement](#)
- [_START_TRANSACTION_statement](#)
- [_SUBQUERY_substatement](#)
- [_TRUNCATE_TABLE_statement](#)
- [_UNION_set_operator](#)
- [_UPDATE_statement](#)
- [_web_site](#)
- [_WHERE_clause](#)

PostgreSQL keywords

pound symbol (#)

POW function

- [_MySQL](#)
- [_PostgreSQL](#)

POWER function

- [_MySQL](#)

POWERMULTISET function (Oracle)

prefixes

PreparedStatement set methods (JDBC)

PRESENTNNV function (Oracle)

PRESENTV function (Oracle)

PREVIOUS function (Oracle)

primary databases (Oracle)

PRIMARY_KEY constraints

PROC_ROLE function

procedural programming

PROCEDURE, quick reference

PROCESSADMIN (SQL Server system role)

programming databases

- [_examples](#)
- [_overview](#)





< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

QUARTER function

[_DB2](#)

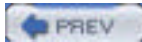
[_MySQL](#)

[quoted_identifier_symbol](#)

[quoted_identifiers](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[RADIANS function](#)

[_DB2](#)

[_MySQL](#)

[_PostgreSQL](#)

[_SQL Server](#)

[RADIUS function \(PostgreSQL\)](#)

[RAISE_ERROR function \(DB2\)](#)

[RAND function](#)

[_DB2](#)

[_SQL Server](#)

[RAND\(\) function \(MySQL\)](#)

[RANK function 2nd](#)

[RANK\(\) function](#)

[RATIO_TO_REPORT function \(Oracle\)](#)

[RAW datatype](#)

[_Oracle](#)

[_platform comparison](#)

[RAWTOHEX function \(Oracle\)](#)

[RDBMSs \(Relational Database Management System\)](#)

[ReadCommitted isolation level \(Transaction object \(ADO.NET\)\)](#)

[ReadUncommitted isolation level \(Transaction object \(ADO.NET\)\)](#)

[REAL datatype](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[REAL function \(DB2\)](#)

[REC2XML function \(DB2\)](#)

[records](#)

[REF function \(Oracle\)](#)

[REFTOHEX function \(Oracle\)](#)

[REGEXP_INSTR function \(Oracle\)](#)

[REGEXP_REPLACE function \(Oracle\)](#)

[REGEXP_SUBSTR function \(Oracle\)](#)

[REGR family of functions](#)

[REGR_AVGX function 2nd 3rd 4th 5th 6th 7th 8th 9th 10th](#)

[REGR_AVGY function 2nd 3rd 4th 5th 6th 7th 8th 9th](#)

[REGR_COUNT function 2nd 3rd 4th 5th 6th 7th 8th](#)

[REGR_INTERCEPT function 2nd 3rd 4th 5th 6th 7th](#)

[REGR_R2 function 2nd 3rd 4th 5th 6th](#)

[REGR_SLOPE function 2nd 3rd 4th 5th](#)

[REGR_SXX function 2nd 3rd 4th](#)

[REGR_SXY function 2nd 3rd](#)

[REGR_SYY function 2nd](#)

[Relational Database Management Systems \(RDBMSs\)](#)

[relational databases, principles of](#)

[RELEASE SAVEPOINT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[RELEASE_LOCK function \(MySQL\)](#)

[RELTIME function \(PostgreSQL\)](#)

[REMAINDER function \(Oracle\)](#)

[REPEAT function](#)

- [_DB2](#)
- [_MySQL](#)

[RepeatableRead isolation level \(Transaction object \(ADO.NET\)\)](#)

[REPLACE function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_SQL Server](#)

[REPLICATE function](#)

- [_SQL Server](#)

[reserved words 2nd](#)

[resources, freeing](#)

[ResultSet methods \(JDBC\)](#)

- [_Get](#)

[RETURN statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[REVERSE function](#)

- [_MySQL](#)
- [_SQL Server](#)

[REVOKE statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[RIGHT \[OUTER\] JOIN](#)

[RIGHT function](#)

- [_DB2](#)
- [_MySQL](#)
- [_SQL Server](#)

[ROLE_CONTAIN function](#)

[ROLE_ID function](#)

[ROLE_NAME function](#)

[ROLLBACK statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[ROUND function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[row processing](#)

[ROW_NUMBER function](#)

[ROWCNT function](#)

[ROWCOUNT_BIG\(.\) function](#)

- [_SQL Server](#)

[ROWID datatype](#)

- [_Oracle](#)
- [_platform comparison](#)

[ROWIDTOCHAR function \(Oracle\)](#)

[rows, filtering and comparison operators](#)

[ROWVERSION datatype](#)

- [_platform comparison](#)
- [_SQL Server 2nd](#)

[RPAD function](#)

- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)

[RTRIM function](#)

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL Server](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[sampling](#)

[SAVEPOINT statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[scalar functions 2nd](#)

[_built-in](#)

[_categories of](#)

[scalar subqueries](#)

[Scalar subquery in SELECT item list](#)

[Scalar subquery in WHERE/HAVING clause](#)

[schema addressing](#)

[schemas](#)

[SCN_TO_TIMESTAMP function \(Oracle\)](#)

[SEC_TO_TIME function \(MySQL\)](#)

[SECOND function](#)

[_DB2](#)

[_MySQL](#)

[SECURITYADMIN \(SQL Server system role\)](#)

[SELECT statement](#)

[_DB2](#)

[_examples](#)

[_GROUP BY clause](#)

[_HAVING clause](#)

[_keywords](#)

[_ALL](#)

[_AS](#)

[_DISTINCT](#)

[_FROM](#)

[_GROUP BY](#)

[_HAVING](#)

[_JOIN](#)

[_ORDER BY](#)

[_WHERE](#)

[_MySQL](#)

[_operators and 2nd](#)

[_Oracle](#)

[_ORDER BY clause](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_quick reference](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[SEQUEL \(Structured English Query Language\)](#)

SERIAL datatype

- [_platform_comparison](#)
- [_PostgreSQL](#)

SERIAL4 datatype

- [_platform_comparison](#)
- [_PostgreSQL](#)

SERIAL8 datatype

- [_platform_comparison](#)
- [_PostgreSQL](#)

[Serializable isolation level \(Transaction object \(ADO.NET\)\)](#)

[SERVERADMIN \(SQL Server system role\)](#)

[SESSION_USER function](#)

[SESSION_USER\(\) function \(MySQL\)](#)

[SESSIONTIMEZONE function \(Oracle\)](#)

[SET CATALOG statement, quick reference](#)

[SET COLLATION statement, quick reference](#)

[SET CONNECTION statement](#)

- [_DB2](#)
- [_description](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET CONSTRAINT statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

SET datatype

- [_MySQL](#)
- [_platform_comparison](#)

[SET DESCRIPTOR statement, quick reference](#)

[SET function \(Oracle\)](#)

[set methods \(JDBC PreparedStatement\)](#)

[set operations](#)

[SET PATH statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET ROLE statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET SCHEMA statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET SESSION AUTHORIZATION statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET TIME_ZONE statement](#)

- [_DB2](#)
- [_keywords](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET TRANSACTION statement](#)

- [_DB2](#)
- [_keywords](#)

- [_MySQL](#)
- [_Oracle](#)
- [_platform support for](#)
- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)

[SET_MASKLEN function \(PostgreSQL\)](#)

[SETUPADMIN \(SQL Server system role\)](#)

[SHA function \(MySQL\)](#)

[SHA1 function \(MySQL\)](#)

[shared keywords](#)

[SHOW_ROLE\(\) function](#)

[SHOW_SEC_SERVICES\(\) function](#)

SIGN function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_SQL Server](#)

SIN function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[Single parameter object type \(ADO.NET\)](#)

SINH function

- [_DB2](#)
- [_Oracle](#)

SMALLDATETIME datatype

- [_platform comparison](#)
- [_SQL Server 2nd](#)
- [_Sybase adaptive server](#)

SMALLINT datatype

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_platform comparison](#)
- [_PostgreSQL](#)
- [_SQL Server](#)

[SMALLINT function \(DB2\)](#)

SMALLMONEY datatype

- [_platform comparison](#)
- [_SQL Server](#)

SOME operator 2nd

- [_keywords](#)
- [_platform differences](#)
- [_platform support for](#)
- [_programming tips](#)
- [_quick reference](#)
- [_rules](#)
- [_SQL2003 syntax](#)

[SORTKEY function](#)

SOUNDEX function

- [_DB2](#)
- [_MySQL](#)
- [_Oracle](#)
- [_SQL Server](#)

[SPACE function](#)

[_DB2](#)

[_MySQL](#)

[_SQL_Server](#)

[SQL command reference, quick reference](#)

[SQL dialects](#)

[SQL history and implementations](#)

[SQL Server](#)

[_allowed_characters_in_identifiers](#)

[_basics](#)

[_beginning_identifier_character](#)

[_BIGINT_datatype](#)

[_BINARY_datatype](#)

[_CALL_statement](#)

[_CHAR\(\)_datatype](#)

[_CLOSE_CURSOR_statement](#)

[_COMMIT_statement](#)

[_CONNECT_statement](#)

[_connecting_using_JDBC](#)

[_CREATE/ALTER_DATABASE_statements](#)

[_ADD_FILEGROUP](#)

[_ADD_LOG](#)

[_ANSI_NULL_DEFAULT](#)

[_ANSI_NULLS](#)

[_ANSI_PADDING](#)

[_ANSI_WARNINGS](#)

[_ARITHABORT](#)

[_AUTO_CLOSE](#)

[_AUTO_CREATE_STATISTICS](#)

[_auto_option](#)

[_AUTO_SHRINK](#)

[_AUTO_UPDATE_STATISTICS](#)

[_COLLATE](#)

[_CONCAT_NULL_YIELDS_NULL](#)

[_CURSOR_CLOSE_ON_COMMIT](#)

[_CURSOR_DEFAULT](#)

[_FOR_LOAD](#)

[_MODIFY_FILE](#)

[_MODIFY_FILEGROUP](#)

[_MODIFY_NAME](#)

[_MULTI_USER](#)

[_NUMERIC_ROUNDABORT](#)

[_OFFLINE](#)

[_ON_file_definition](#)

[_QUOTED_IDENTIFIER](#)

[_READ_ONLY](#)

[_RECOVERY](#)

[_RECURSIVE_TRIGGERS](#)

[_REMOVE_FILE](#)

[_REMOVE_FILEGROUP](#)

[_RESTRICTED_USER](#)

[_SET](#)

[_SINGLE_USER](#)

[_sql_options_clause](#)

[_TORN_PAGE_DETECTION](#)

[_WITH](#)

[_CREATE/ALTER_FUNCTION/PROCEDURE_statements](#)

[_@parameter_datatype](#)

[_CREATE](#)

- ENCRYPTION
- FOR REPLICATION
- RECOMPILE
- RETURNS
- SCHEMABINDING
- WITH
- CREATE/ALTER INDEX statements
 - ASC
 - CLUSTERED
 - DESC
 - DROP_EXISTING
 - FILLFACTOR
 - IGNORE_DUP_KEY
 - ON
 - PAD_INDEX
 - SORT_IN_TEMPDB
 - STATISTICS_NORECOMPUTE
 - WITH
- CREATE/ALTER METHOD statements
- CREATE/ALTER ROLE statements
- CREATE/ALTER SCHEMA statements
- CREATE/ALTER TABLE statements
 - CHECK
 - CHECK CONSTRAINT
 - COLLATE
 - DEFAULT
 - ENABLE TRIGGER
 - FOREIGN KEY
 - IDENTITY
 - NOT FOR REPLICATION
 - PRIMARY KEY
 - ROWGUIDCOL
 - TEXTIMAGE_ON
 - WITH CHECK
- CREATE/ALTER TRIGGER statements
- CREATE/ALTER TYPE statements
- CREATE/ALTER VIEW statements
- datatypes
 - compared to Sybase adaptive server
- DATETIME datatype
- DECLARE CURSOR command
 - DYNAMIC
 - FAST_FORWARD
 - FOR
 - FOR UPDATE
 - FORWARD_ONLY
 - GLOBAL
 - INSENSITIVE
 - KEYSET
 - LOCAL
 - OPTIMISTIC
 - READ_ONLY
 - SCROLL
 - SCROLL_LOCKS
 - SQL-92 compatibility syntax
 - STATIC
 - Transact-SQL extensioned syntax
 - TYPE_WARNING
- DELETE statement

[delimiters and operators](#)

[DISCONNECT statement](#)

[DROP statements](#)

[DATABASE](#)

[DEFAULT](#)

[FUNCTION](#)

[INDEX](#)

[PROCEDURE](#)

[RULE](#)

[STATISTICS](#)

[TABLE](#)

[TRIGGER](#)

[TYPE](#)

[VIEW](#)

[EXCEPT set operator](#)

[FETCH statement](#)

[functions](#)

[ACOS](#)

[APP_NAME](#)

[ASCII](#)

[ASIN](#)

[ATAN](#)

[ATN2](#)

[BINARY_CHECKSUM](#)

[CHAR](#)

[CHARINDEX](#)

[CHECKSUM](#)

[CHECKSUM_AGG](#)

[COALESCE](#)

[COL_LENGTH](#)

[COL_NAME](#)

[CONTAINS](#)

[CONTAINSTABLE](#)

[CONVERT](#)

[COS](#)

[COT](#)

[DATABASEPROPERTYEX](#)

[DATALENGTH](#)

[DATEADD](#)

[DATEDIFF](#)

[DATENAME](#)

[DATEPART](#)

[DAY](#)

[DB_ID](#)

[DB_NAME](#)

[DEGREES](#)

[DIFFERENCE](#)

[FILE_ID](#)

[FILE_NAME](#)

[FILEGROUP_ID](#)

[FILEGROUP_NAME](#)

[FILEPROPERTY](#)

[FORMATMESSAGE](#)

[FREETEXT](#)

[FREETEXTTABLE](#)

[FULLTEXTCATALOGPROPERTY](#)

[FULLTEXTSERVICEPROPERTY](#)

[GETANSINULL](#)

[GETDATE\(.\)](#)

GETUTCDATE()
GROUPING
HOST_ID()
HOST_NAME()
IDENT_CURRENT
IDENT_INCR
IDENT_SEED
IDENTITY
INDEX_COL
INDEXPROPERTY
IS_MEMBER
IS_SRVROLEMEMBER
ISDATE
ISNULL
ISNUMERIC
LEFT
LEN
LOG
LOG10
LTRIM
MONTH
NCHAR
NEWID()
NULLIF
OBJECT_ID
OBJECT_NAME
OBJECTPROPERTY
OPEN
OPENDATASOURCE
OPENQUERY
OPENROWSET
PARSENAME
PATINDEX
PERMISSIONS
PI()
RADIANS
RAND
REPLACE
REPLICATE
REVERSE
RIGHT
ROUND
ROWCOUNT_BIG
RTRIM
SIGN
SIN
SOUNDEX
SPACE
STATS_DATE
STDEV
STDEVPI
STR
STUFF
SUBSTRING
SUSER_ID
SUSER_SID
SUSER_SNAME
TAN
TEXTPTR

- [_TEXTVALID](#)
- [_TYPEPROPERTY](#)
- [_UNICODE](#)
- [_USER_ID](#)
- [_USER_NAME](#)
- [_VAR](#)
- [_VARP](#)
- [_YEAR](#)
- [_GRANT statement](#)
- [_identifier size](#)
- [_INSERT statement](#)
- [_INTERSECT set operator](#)
- [_JOIN subclause](#)
- [_keywords](#)
- [_LIKE operator](#)
- [_MERGE statement](#)
- [_NCHAR\(\) datatype 2nd](#)
- [_nested triggers](#)
- [_NTEXT datatype](#)
- [_OPEN statement](#)
- [_ORDER BY clause](#)
- [_recursive triggers](#)
- [_RELEASE SAVEPOINT statement](#)
- [_reserved identifiers](#)
- [_RETURN statement](#)
- [_REVOKE statement](#)
- [_ROLLBACK statement](#)
- [_ROWVERSION datatype](#)
- [_SAVEPOINT statement](#)
- [_schema addressing](#)
- [_SELECT statement](#)
 - [_COMPUTE clause](#)
 - [_FOR](#)
 - [_FROM](#)
 - [_GROUP BY](#)
 - [_INTO](#)
 - [_OPTION](#)
 - [_ORDER BY](#)
 - [_TOP](#)
- [_SET CONNECTION statement](#)
- [_SET CONSTRAINT statement](#)
- [_SET PATH statement](#)
- [_SET ROLE statement](#)
- [_SET SCHEMA statement](#)
- [_SET SESSION AUTHORIZATION statement](#)
- [_SET statement](#)
- [_SET TIME ZONE statement](#)
- [_SET TRANSACTION statement](#)
- [_SMALLDATETIME datatype](#)
- [_SQL_VARIANT datatype](#)
- [_START TRANSACTION statement](#)
- [_stored procedures 2nd](#)
- [_SUBQUERY substatement](#)
- [_system roles](#)
 - [_BULKADMIN](#)
 - [_DB_ACCESSADMIN](#)
 - [_DB_BACKUPOPERATOR](#)
 - [_DB_DATAREADER](#)
 - [_DB_DATAWRITER](#)

- [_DB_DDLADMIN](#)
- [_DB_DENYDATAREADER](#)
- [_DB_DENYDATAWRITER](#)
- [_DB_OWNER](#)
- [_DB_SECURITYADMIN](#)
- [_DBCREATOR](#)
- [_DISKADMIN](#)
- [_PROCESSADMIN](#)
- [_SECURITYADMIN](#)
- [_SERVERADMIN](#)
- [_SETUPADMIN](#)
- [_SYSADMIN](#)
- [_TRUNCATE_TABLE statement](#)
- [_UDFs](#)
- [_UNION set operator](#)
- [_UNIQUEIDENTIFIER datatype](#)
- [_UPDATE statement](#)
- [_VARBINARY\(\) datatype](#)
- [_VARCHAR\(\) datatype](#)
- [_versus Sybase adaptive server](#)
 - [_functions](#)
- [_web site](#)
- [_WHERE clause](#)
- [SQL Standard web site](#)
- [SQL/CLI \(Call-Level Interface\)](#)
- [SQL/Foundation](#)
- [SQL/Framework](#)
- [SQL/JRT \(Java Routines and Types\)](#)
- [SQL/MED \(Management of External Data\)](#)
- [SQL/OBJ \(Object Language Binding\)](#)
- [SQL/PSM \(Persistent Stored Modules\)](#)
- [SQL/Schemata](#)
- [SQL/XML 2nd](#)
- [SQL_VARIANT datatype](#)
 - [_platform comparison](#)
 - [_SQL Server 2nd](#)
- [SQL2003](#)
 - [_allowed characters in identifiers](#)
 - [_beginning identifier character](#)
 - [_categories and datatypes](#)
 - [_data structures](#)
 - [_identifier size](#)
 - [_keywords](#)
 - [_new features](#)
 - [_numeric functions](#)
 - [_reserved identifiers](#)
 - [_schema addressing](#)
 - [_statement classes](#)
 - [_supplemental features packages](#)
 - [_window syntax](#)
- [SQL2003 keywords](#)
- [SQL2003 syntax](#)
 - [_ALL operator](#)
 - [_ANY operator](#)
 - [_BETWEEN operator](#)
 - [_CALL statement](#)
 - [_CLOSE CURSOR statement](#)
 - [_COMMIT statement](#)
 - [_CONNECT statement](#)

[_CREATE/ALTER FUNCTION/PROCEDURE statements](#)
[_CREATE/ALTER METHOD statements](#)
[_CREATE/ALTER ROLE statements](#)
[_CREATE/ALTER SCHEMA statements](#)
[_CREATE/ALTER TABLE statements](#)
[_CREATE/ALTER TRIGGER statements](#)
[_CREATE/ALTER TYPE statements](#)
[_CREATE/ALTER VIEW statements](#)
[_DECLARE CURSOR command](#)
[_DELETE statement](#)
[_DISCONNECT statement](#)
[_DROP statements](#)
[_EXCEPT set operator](#)
[_EXISTS operator](#)
[_FETCH statement](#)
[_GRANT statement](#)
[_IN operator](#)
[_INSERT statement](#)
[_INTERSECT set operator](#)
[_IS operator](#)
[_JOIN subclause](#)
[_LIKE operator](#)
[_MERGE statement](#)
[_OPEN statement](#)
[_ORDER BY clause](#)
[_RELEASE SAVEPOINT statement](#)
[_RETURN statement](#)
[_REVOKE statement](#)
[_ROLLBACK statement](#)
[_SAVEPOINT statement](#)
[_SELECT statement](#)
[_SET CONNECTION statement](#)
[_SET CONSTRAINT statement](#)
[_SET PATH statement](#)
[_SET ROLE statement](#)
[_SET SCHEMA statement](#)
[_SET SESSION AUTHORIZATION statement](#)
[_SET statement](#)
[_SET TIME ZONE statement](#)
[_SET TRANSACTION statement](#)
[_SOME operator](#)
[_START TRANSACTION statement](#)
[_SUBQUERY substatement](#)
[_UNION set operator](#)
[_UPDATE statement](#)
[_WHERE clause](#)

[SQL92](#)
[SQL99](#)
[_deleted elements in SQL2003](#)

[SqlConnection](#)
[_connection string attributes for](#)

[SQLPL](#)
[SQRT function](#)
[standby databases \(Oracle\)](#)
[START TRANSACTION statement](#)
[_DB2](#)
[_MySQL](#)
[_Oracle](#)
[_platform support for](#)

- [_PostgreSQL](#)
- [_programming tips](#)
- [_rules](#)
- [_SQL Server](#)
- [_SQL2003 syntax](#)
- statement objects
 - [_associating SQL with](#)
 - [_creating](#)
 - [_executing](#)
- statements
 - [_executing](#)
 - [_preparing and executing](#)
 - [_re-executing](#)
 - [_Sybase adaptive server SQL](#)
- [STATS_BINOMIAL_TEST function \(Oracle\)](#)
- [STATS_DATE function](#)
 - [_SQL Server](#)
- [STD function \(MySQL\)](#)
- [STDDEV function](#)
 - [_DB2](#)
 - [_MySQL](#)
 - [_Oracle](#)
- [STDDEV_POP function 2nd](#)
- [STDDEV_SAMP function 2nd](#)
 - [_Oracle](#)
- [STDEV function](#)
 - [_SQL Server](#)
- [STDEV_POP function \(Oracle\)](#)
- [STDEV_P function](#)
 - [_SQL Server](#)
- [STOP LOGICAL STANDBY APPLY](#)
- [storage_clause \(Oracle\)](#)
- stored procedures
 - [_operators and 2nd](#)
- [STR function \(SQL Server\)](#)
- [STRCMP function \(MySQL\)](#)
- string functions and operators
- [String parameter object type \(ADO.NET\)](#)
- [StringFixedLength parameter object type \(ADO.NET\)](#)
- [STUFF function \(SQL Server\)](#)
- [SUBDATE function \(MySQL\)](#)
- subpartitioned tables
 - [_Oracle](#)
 - [_altering](#)
- subqueries
 - [_correlated](#)
 - [_nested table](#)
 - [_scalar](#)
 - [_table](#)
- [SUBQUERY substatement](#)
 - [_DB2](#)
 - [_examples](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for 2nd](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_rules](#)

- [_SQL_Server](#)
- [_SQL2003_syntax](#)
- SUBSTR function
 - [_DB2](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
- SUBSTRB function (Oracle)
- SUBSTRING function
 - [_MySQL](#)
 - [_PostgreSQL](#)
 - [_SQL_Server](#)
- SUBSTRING_INDEX function (MySQL)
- subtraction (-) arithmetic operator
- suffixes
- SUM function 2nd
- SUSER_ID function
 - [_SQL_Server](#)
- SUSER_NAME function
- SUSER_SID function
 - [_SQL_Server](#)
- SUSER_SNAME function
 - [_SQL_Server](#)
- SYB_SENDFMSG function
- Sybase Adaptive Aerver
 - [_TIMESTAMP datatype](#)
- Sybase Adaptive Server
 - [_BINARY datatype](#)
 - [_CHAR\(\) datatype](#)
 - [_datatypes](#)
 - [_compared to SQL_Server](#)
 - [_DATETIME datatype](#)
 - [_keywords 2nd](#)
 - [_NCHAR\(\) datatype](#)
 - [_NVARCHAR\(\) datatype](#)
 - [_SMALLDATETIME datatype](#)
 - [_SQL statements](#)
 - [_statements](#)
 - [_supported functions](#)
 - [_UNICHAR datatype](#)
 - [_UNIVARCHAR datatype](#)
 - [_VARBINARY\(\) datatype](#)
 - [_VARCHAR\(\) datatype](#)
 - [_versus SQL_Server](#)
 - [_functions](#)
- Sybase Adaptive Server keywords
- Sybase adaptive server keywords
- SYS_CONNECT_BY_PATH function (Oracle)
- SYS_CONTEXT function (Oracle)
- SYS_DBURIGEN function (Oracle)
- SYS_EXTRACT_UTC function (Oracle)
- SYS_GUID() function (Oracle)
- SYS_TYPEID function (Oracle)
- SYS_XMLAGG function (Oracle)
- SYS_XMLGEN function (Oracle)
- SYSADMIN (SQL Server system role)
- SYSDATE function (Oracle)
- SYSDATE() function (MySQL)
- system_delimiters
- SYSTEM_USER function

[SYSTEM_USER\(.\) function \(MySQL\)](#)

[SYSTIMESTAMP function \(Oracle\)](#)

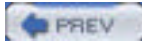


< Day Day Up >



[**SYMBOL**] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

/ division arithmetic operator
!> not greater than operator
!< not less than operator
!≡ not equal to operator
(pound symbol)
(double-pound symbol)
% modula arithmetic operator
& bitwise AND operator
> greater than operator
>= greater than or equal to operator
< less than operator
<= less than or equal to operator
* multiplication arithmetic operator
+ addition arithmetic operator
+ unary operator
- subtraction arithmetic operator
- unary operator
≡ assignment operator
≡ equal to comparison operator
^ bitwise exclusive OR operator
| bitwise OR operator
~ unary operator





[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

TABLE datatype

[_platform_comparison](#)

[_SQL_Server](#)

table_subqueries

[_nested](#)

table-level constraints

[TABLE_NAME function \(DB2\)](#)

[TABLE_SCHEMA function \(DB2\)](#)

tables

[TABLESAMPLE clause](#)

TAN function

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL_Server](#)

TANH function

[_DB2](#)

[_Oracle](#)

[testing_expressions_against_range_of_values](#)

TEXT datatype

[_MySQL](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[TEXT function \(PostgreSQL\)](#)

[TEXTPTR function \(SQL Server\)](#)

[TEXTVALID function \(SQL Server\)](#)

TIME datatype

[_DB2](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[TIME function \(DB2\)](#)

[TIME_FORMAT function \(MySQL\)](#)

[TIME_TO_SEC function \(MySQL\)](#)

TIMESPAN datatype

[_platform_comparison](#)

[_PostgreSQL](#)

TIMESTAMP datatype

[_DB2](#)

[_Oracle](#)

[_platform_comparison](#)

[_PostgreSQL](#)

[_SQL_Server](#)

[_Sybase_adaptive_server](#)

TIMESTAMP function

[_DB2](#)

[_PostgreSQL](#)

[TIMESTAMP_FORMAT function \(DB2\)](#)

[TIMESTAMP_ISO function \(DB2\)](#)

[TIMESTAMP_TO_SCN function \(Oracle\)](#)

[TIMESTAMPDIFF function \(DB2\)](#)

TIMETZ datatype

[_platform_comparison](#)

- [_PostgreSQL](#)
- TINYINT datatype
 - [_platform_comparison](#)
 - [_SQL_Server](#)
- [TO_BINARY_DOUBLE function \(Oracle\)](#)
- [TO_BINARY_FLOAT function \(Oracle\)](#)
- TO_CHAR function
 - [_DB2](#)
 - [_Oracle_2nd](#)
 - [_PostgreSQL](#)
- [TO_CLOB function \(Oracle\)](#)
- TO_DATE function
 - [_DB2](#)
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [TO_DAYS function \(MySQL\)](#)
- [TO_DSINTERVAL function \(Oracle\)](#)
- [TO_LOB function \(Oracle\)](#)
- [TO_MULTI_BYTE function \(Oracle\)](#)
- [TO_NCHAR function \(Oracle\)](#)
- [TO_NCLOB function \(Oracle\)](#)
- TO_NUMBER function
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [TO_SINGLE_BYTE function \(Oracle\)](#)
- TO_TIMESTAMP function
 - [_Oracle](#)
 - [_PostgreSQL](#)
- [TO_TIMESTAMP_TZ function \(Oracle\)](#)
- [TO_UNICHAR function](#)
- [TO_YMINTERVAL function \(Oracle\)](#)
- Transact-SQL
- Transaction object
 - [_ADO.NET](#)
 - [_JDBC](#)
- [TRANSACTION_NONE isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_READ_COMMITTED isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_READ_UNCOMMITTED isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_REPEATABLE_READ isolation level \(Transaction object \(JDBC\)\)](#)
- [TRANSACTION_SERIALIZABLE isolation level \(Transaction object \(JDBC\)\)](#)
- transactions
 - [_beginning_2nd](#)
 - [_ADO.NET](#)
 - [_JDBC](#)
 - [_closing_and_making_permanent](#)
 - [_committing](#)
 - [_ending](#)
 - [_explicit](#)
 - [_implicit](#)
 - [_managing](#)
 - [_rolling_back](#)
- TRANSLATE function
 - [_DB2](#)
 - [_Oracle_2nd](#)
 - [_PostgreSQL](#)
- [TREAT function \(Oracle\)](#)
- [TRIM function](#)
- TRUNC function
 - [_DB2](#)

- [_Oracle 2nd](#)
- [_PostgreSQL](#)
- TRUNCATE function
 - [_DB2](#)
 - [_MySQL](#)
- TRUNCATE TABLE statement
 - [_DB2](#)
 - [_defacto standard syntax](#)
 - [_keywords](#)
 - [_MySQL](#)
 - [_Oracle](#)
 - [_platform support for](#)
 - [_PostgreSQL](#)
 - [_programming tips](#)
 - [_rules](#)
 - [_SQL Server](#)
- TSEQUEL function
- Twelve Principles of Relational Databases
- TYPE_ID function (DB2)
- TYPE_NAME function (DB2)
- TYPE_SCHEMA function (DB2)
- TYPEPROPERTY function
 - [_SQL Server](#)
- TZ_OFFSET function (Oracle)





[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [**U**] [V] [W] [X] [Y]

[UCASE function \(MySQL\)](#)

[UHIGHSURR function](#)

[UID function \(Oracle\)](#)

[UInt parameter object type \(ADO.NET\)](#)

[ULOWSURR function](#)

[unary operators 2nd](#)

[UNCOMPRESS function \(MySQL\)](#)

[UNHEX function \(MySQL\)](#)

[UNICHAR datatype \(Sybase adaptive server\)](#)

[UNICODE function](#)

[_SQL Server](#)

[UNION set operator](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[UNIQUE constraints](#)

[UNIQUEIDENTIFIER datatype](#)

[_platform comparison](#)

[_SQL Server 2nd](#)

[UNISTR function \(Oracle\)](#)

[UNIVARCHAR datatype \(Sybase adaptive server\)](#)

[UNIX_TIMESTAMP function \(MySQL\)](#)

[Unspecified isolation level \(Transaction object \(ADO.NET\)\)](#)

[UPDATE statement](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_operators and 2nd](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

[UPDATEXML function \(Oracle\)](#)

[UPPER function](#)

[URLs](#)

[_MySQL](#)

[_open source](#)

[_Oracle](#)

[_PostgreSQL](#)

[_SQL Server](#)

[UROWID datatype](#)

[_Oracle](#)

[_platform comparison](#)

[USCALAR function](#)

[user](#)

[USER\(.\) function \(MySQL\)](#)

[USER_ID function \(SQL Server\)](#)

[USER_NAME function \(SQL Server\)](#)

[USERENV function \(Oracle\)](#)



< Day Day Up >



[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[VALID_NAME function](#)

[VALID_USER function](#)

[VALUE function](#)

[_DB2](#)

[_Oracle](#)

[VAR function](#)

[_DB2](#)

[_SQL Server](#)

[VAR_POP function 2nd](#)

[_Oracle](#)

[VAR_SAMP function 2nd](#)

[_Oracle](#)

[VARBINARY datatype](#)

[_platform comparison](#)

[_SQL Server](#)

[VARBINARY\(\) datatype](#)

[_SQL Server](#)

[_Sybase adaptive server](#)

[VARBIT datatype](#)

[_platform comparison](#)

[VARCHAR datatype](#)

[_DB2](#)

[_MySQL](#)

[_Oracle](#)

[_platform comparison](#)

[_PostgreSQL](#)

[_SQL Server](#)

[VARCHAR FOR BIT DATA datatype](#)

[_DB2](#)

[_platform comparison](#)

[VARCHAR function](#)

[_DB2](#)

[_PostgreSQL](#)

[VARCHAR\(\) datatype](#)

[_SQL Server](#)

[_Sybase adaptive server](#)

[VARCHAR_FORMAT function \(DB2\)](#)

[VARCHAR2 datatype](#)

[_Oracle](#)

[_platform comparison](#)

[VARGRAPHIC datatype](#)

[_DB2](#)

[_platform comparison](#)

[VARGRAPHIC function \(DB2\)](#)

[VARIANCE function](#)

[_DB2](#)

[_Oracle](#)

[VARP function](#)

[_SQL Server](#)

[Vector subquery in WHERE/HAVING clause](#)

[VERSION\(\) function \(MySQL\)](#)

[views 2nd](#)

[_operators and 2nd](#)

[VSIZE function \(Oracle\)](#)



[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Y\]](#)

WEEK function

[_DB2](#)

[_MySQL](#)

[WEEK_ISO function \(DB2\)](#)

[WEEKDAY function \(MySQL\)](#)

[WHERE clause 2nd 3rd](#)

[_DB2](#)

[_keywords](#)

[_MySQL](#)

[_Oracle](#)

[_platform support for](#)

[_PostgreSQL](#)

[_programming tips](#)

[_rules](#)

[_SQL Server](#)

[_SQL2003 syntax](#)

WHERE clause (SELECT statement)

[_comparison operators and](#)

[_logical operators and](#)

[WIDTH function \(PostgreSQL\)](#)

[WIDTH_BUCKET function](#)

[window functions 2nd](#)

[_DB2 syntax](#)

[_group clause](#)

[_Oracle syntax](#)

[_ordering clause](#)

[_partitioning clause](#)

[_SQL2003 syntax](#)

[windowing functions](#)

[WITH\[OUT\] SESSION SHUTDOWN](#)



[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[Xbase](#)

XML datatype

[_Oracle](#)

[_platform_comparison](#)

XML schema

[_tables_based_on](#)

[XMLAGG function \(Oracle\)](#)

[XMLCOLATTVAL function \(Oracle\)](#)

[XMLCONCAT function \(Oracle\)](#)

[XMLELEMENT function \(Oracle\)](#)

[XMLFOREST function \(Oracle\)](#)

[XMLSEQUENCE function \(Oracle\)](#)

[XMLTRANSFORM function \(Oracle\)](#)





< Day Day Up >

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]

[YEAR function](#)

[_DB2](#)

[_MySQL](#)

[_SQL Server](#)

[YEARWEEK function \(MySQL\)](#)



< Day Day Up >



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)
- [Academic](#)

SQL in a Nutshell, 2nd Edition

By [Kevin E. Kline](#)

Publisher: O'Reilly

Pub Date: September 2004

ISBN: 0-596-00481-8

Pages: 710

This essential desktop reference drills down and documents every SQL command and how to use it in both commercial (Oracle, DB2, and Microsoft SQL Server) and open source implementations (PostgreSQL, and MySQL). It includes the command syntax (by vendor, if the syntax differs across implementations), a description, and practical examples. And it also explains how the leading commercial and open sources database product implement SQL. This wealth of information is packed into a succinct, comprehensive, and extraordinarily easy-to-use format.

A.1 Sybase Adaptive Server Datatypes

[Table A-1](#) compares the datatypes supported by Sybase Adaptive Server to those supported by Microsoft SQL Server. While there is significant overlap in the types offered, a few types differ in meaning or are missing on one of the platforms. Where differences in the types exist, they'll be listed in the table. Types that are not listed within the table should be considered identical on both platforms.

Table A-1. Sybase Adaptive Server datatypes

Sybase adaptive server datatype	Microsoft SQL Server datatype	Notes
N/A	<i>BIGINT</i>	Usage of <i>decimal(19,0)</i> as an equivalent to <i>BIGINT</i> will allow storage of values within the same range.
<i>BINARY</i>	<i>BINARY</i>	Describes a fixed-length binary value up to 255 bytes in size. In contrast, Microsoft SQL Server can store 8,000 bytes.
<i>CHAR(N)</i>	<i>CHAR(N)</i>	Holds fixed-length character data up to the database's page size in length.
<i>DATETIME</i>	<i>DATETIME</i>	Holds date and time data within the range of 1753-01-01 00:00:00 through 9999-12-31 23:59:59. The <i>DATETIME</i> type can store milliseconds with a granularity of 1/300 second.
<i>NCHAR(N)</i>	N/A	Holds fixed-length character strings in the national character set up to the database's page size in length. In contrast, Microsoft SQL Server's <i>NCHAR</i> datatype stores Unicode character strings.
N/A	<i>NTEXT</i>	Holds Unicode text passages up to 1,073,741,823 characters in length.
<i>NVARCHAR(N)</i>	N/A	Holds variable-length character strings in the national character set up to the database's page size in length. In contrast, Microsoft SQL Server's <i>NVARCHAR</i> datatype stores Unicode character strings.
<i>TIMESTAMP</i>	<i>ROWVERSION</i>	A unique number within a database that is updated whenever a row is updated.
<i>SMALLDATETIME</i>	<i>SMALLDATETIME</i>	Holds date and time data within the range of 1900-01-01 00:00 through 2079-12-31 23:59. Unlike Microsoft SQL Server, Sybase cannot hold dates with second resolution in a <i>SMALLDATETIME</i> .
N/A	<i>SQL_VARIANT</i>	Sybase has no weakly typed storage type like SQL Server's <i>SQL_VARIANT</i> .
<i>UNICHAR</i>	<i>NCHAR(N)</i>	Stores fixed-length, Unicode character strings up to the database's page size in length.
N/A	<i>UNIQUEIDENTIFIER</i>	Represents a value that is unique across all databases and all servers.
<i>UNIVARCHAR</i>	<i>NCHAR(N)</i>	Stores variable-length, Unicode character strings up to the database's page size in length.
<i>VARBINARY(N)</i>	<i>VARBINARY(N)</i>	Describes a variable-length binary value up to 255 bytes in size. In contrast, Microsoft SQL Server can store up to 8,000 bytes.
<i>VARCHAR(N)</i>	<i>VARCHAR(N)</i>	Holds fixed-length character data up to the database's page size in length.

The primary differences between the types offered by SQL Server and Sybase are the storage sizes, precision, and default meaning of the *NCHAR* type. When converting data from one platform to the other, it is important to observe these subtle differences to avoid data truncation or corruption.

A.2 Sybase Adaptive Server SQL Statements

As we saw in the previous section, there is significant overlap in the type support offered by Sybase and SQL Server. This section compares the SQL dialects of Sybase to SQL Server. In most cases, the SQL Server statements described in [Chapter 3](#) apply to Sybase as well. This section outlines those statements that are different between the two platforms.

In [Table A-2](#), the SQL commands are listed along with notes containing the differences between SQL Server and Sybase. While most of the statements have identical syntax and meaning, for those statements that contain large differences, the syntax for Sybase is listed with notes on how the statements differ.

Table A-2. Sybase Adaptive Server SQL statements

Command	Notes
<i>ALTER PROCEDURE</i>	Sybase does not support the <i>ENCRYPTION</i> keyword or replication through the <i>FOR REPLICATION</i> keywords.
<i>ALTER TABLE</i>	<p>Sybase does not support the <i>ROWGUIDCOL</i> keyword or the <i>CHECK</i> and <i>NOCHECK</i> keywords for enabling constraints.</p> <p><i>ALTER TABLE table_name</i></p> <p>[<i>ADD {column_name datatype attributes}{,...}</i>]</p> <p> [<i>DROP column_name{,...}</i>]</p> <p> [<i>ADD CONSTRAINT {constraint_name constraint_clause}{,...}</i>]</p> <p> [<i>DROP CONSTRAINT constraint_name{,...}</i>]</p> <p> [<i>MODIFY {column_name data_type [[NOT] NULL]}{,...}</i>]</p> <p> [<i>{ENABLE DISABLE} trigger_name</i>]</p>
<i>ALTER TRIGGER</i>	Sybase has no equivalent to the <i>ALTER TRIGGER</i> statement in SQL Server. For equivalent functionality, drop the existing trigger with a <i>DROP TRIGGER</i> statement and recreate it with the <i>CREATE TRIGGER</i> statement.
<i>ALTER VIEW</i>	Sybase has no equivalent to the <i>ALTER VIEW</i> statement in SQL Server. For equivalent functionality, drop the existing view with a <i>DROP VIEW</i> statement and recreate it with the <i>CREATE VIEW</i> statement.
<i>CALL</i>	Not supported by SQL Server or Sybase. Look to the <i>EXECUTE</i> command for this functionality.
<i>CONNECT</i>	<p>Supported with variations.</p> <p><i>CONNECT TO server_name</i></p>
<i>CREATE DATABASE</i>	<p>Creating a new database in Sybase has the following syntax:</p> <p><i>CREATE DATABASE database_name</i></p> <p>[<i>ON { DEFAULT database_device} [=size]</i></p> <p> [, { <i>database_device</i> [=size]}{,...}]</p> <p>[<i>LOG ON {log_device [=size]}{,...}</i>]</p> <p>[<i>WITH {OVERRIDE DEFAULT_LOCATION = "pathname"}</i>]</p> <p>[<i>FOR {LOAD PROXY_UPDATE}</i>]</p> <ul style="list-style-type: none"> • <i>DEFAULT</i> specifies the default device, equivalent to the <i>PRIMARY</i> keyword on SQL Server. • <i>size</i> is the size of the database or transaction log in megabytes. • <i>log_device</i> has the same meaning as SQL Server.

	<ul style="list-style-type: none"> • <i>PROXY_UPDATE</i> automatically gets the metadata from the <i>DEFAULT_LOCATION</i> when creating the database.
<i>CREATE FUNCTION</i>	Support is identical to that offered by SQL Server, except that the following options are unsupported: <i>VARYING</i> , <i>ENCRYPTION</i> , and <i>FOR REPLICATION</i> .
<i>CREATE INDEX</i>	<p>Differences exist within the options provided in the <i>WITH</i> clause, as follows:</p> <pre>[WITH [{FILLFACTOR = <i>fillfactor</i> MAX_ROWS_PER_PAGE = <i>num_rows</i>}] [[,]RESERVEPAGEGAP = <i>num_pages</i>] [[,]CONSUMERS = X] [[,]IGNORE_DUP_KEY] [[,]IGNORE_DUP_ROW ALLOW_DUP_ROW] [[,]SORTED_DATA] [[,]STATISTICS USING <i>num_steps</i> VALUES]</pre>
	<ul style="list-style-type: none"> • <i>FILLFACTOR</i> has the same meaning as SQL Server. • <i>MAX_ROWS_PER_PAGE</i> is another way of controlling the fill factor of the index, but uses the row count instead of a percentage of the page space. • <i>RESERVEPAGEGAP</i> provides a method of controlling the ratio of empty index pages to filled ones. This can provide a performance advantage for indexes that grow in size frequently. Valid values for <i>num_pages</i> are 0-255, with the default being 0. • <i>IGNORE_DUP_KEY</i> has the same meaning as SQL Server.
	<ul style="list-style-type: none"> • <i>IGNORE_DUP_ROW</i> allows the creation of a clustered index on tables that contain duplicate rows. While the index can be created when the table contains duplicate rows, any insert or update statement that can result in the creation of a duplicate row will be canceled once the index is created.
<i>CREATE INDEX</i>	<ul style="list-style-type: none"> • <i>ALLOW_DUP_ROW</i> is similar to <i>IGNORE_DUP_ROW</i>, except that it permits insert and update statements that will result in duplicate rows. • <i>SORTED_DATA</i> speeds up the creation of indexes for tables that are already sorted on disk. • <i>WITH STATISTICS USING num_steps VALUES</i> controls the amount of statistics maintained and provided to the query optimizer.
<i>CREATE PROCEDURE</i>	Support is identical to that offered by SQL Server, except that the following options are unsupported: <i>VARYING</i> , <i>ENCRYPTION</i> , and <i>FOR REPLICATION</i> .
<i>CREATE ROLE</i>	<p>Supported with variations.</p> <pre>CREATE ROLE <i>role_name</i> [WITH PASSWD "<i>password</i>" [, {"PASSWD EXPIRATION" "MIN PASSWD LENGTH" "MAX FAILED LOGINS" } <i>option_value</i>][,...]]</pre> <p>This statement will create a role by <i>role_name</i> with an optional password. Unique to Sybase is the ability to specify simple security options for the new role.</p>
	<p>While casual users of SQL Server and Sybase will notice no differences in the CREATE TABLE syntax, others will note the following differences:</p> <ul style="list-style-type: none"> • The NOT FOR REPLICATION option is not supported on Sybase. • The ROWGUIDCOL column attribute is not supported on Sybase.

- The TEXTIMAGE_ON option is not supported on Sybase.
- The IDENTITY attribute cannot have a seed or increment value specified on Sybase.

Other than those small changes, the features unique to Sybase are listed below.

```
CREATE TABLE [database_name.owner].table_name
    ( { column_name datatype
    [ [DEFAULT default_value]
    | [IDENTITY | NULL | NOT NULL]
    | [OFF ROW | IN ROW [ (size_in_bytes) ] ]
    | REFERENCES [[database_name.]owner].ref_table [(ref_column)]
    {UNIQUE | PRIMARY KEY} [CLUSTERED | NONCLUSTERED] [asc |
    desc]
    [WITH { FILLFACTOR = pct,
    MAX_ROWS_PER_PAGE = num_rows, }
    RESERVEPAGEGAP = num_pages } ]
    [ON segment_name]
    | CHECK (search_condition)
    }
    [[CONSTRAINT constraint_name]
    | FOREIGN KEY ( { column_name } [, ... ] ) REFERENCES
    [[database_name.]owner].ref_table
    [ ( { ref_column } [, ... ] ) ]
    | CHECK (search_condition)
    | {UNIQUE | PRIMARY KEY} [CLUSTERED | NONCLUSTERED]
    ( { column_name [ASC | DESC] } [, ... ] )
    [WITH { FILLFACTOR = pct,
    MAX_ROWS_PER_PAGE = num_rows,
    RESERVEPAGEGAP = num_pages } ]
    [ON segment_name] [, ... ] )
    [LOCK {DATAROWS | DATAPAGES | ALLPAGES } ]
    [WITH { MAX_ROWS_PER_PAGE = num_rows,
    EXP_ROW_SIZE = num_bytes,
    RESERVEPAGEGAP = num_pages,
    IDENTITY_GAP = value } ]
    [ON segment_name ]
    [ [ EXTERNAL TABLE ] AT pathname ]
```

CREATE
TABLE

- OFF/IN ROW specifies if a Java-SQL column is physically stored inside a row or outside. The SIZE_IN_BYTES option is the maximum space required to store an IN ROW type.
- ASC/DESC determines the ordering for an index created for a constraint. Ascending order,

CREATE TABLE	<p>ASC, is the default.</p> <ul style="list-style-type: none"> • <i>MAX_ROWS_PER_PAGE</i> limits the number of rows per page. • <i>LOCK</i> specifies the locking strategy used for the table. • <i>EXP_ROW_SIZE</i> specifies the expected row size in bytes. The default is zero, which means the server's default will be used. • <i>RESERVEPAGEGAP</i> specifies the desired ratio of filled to empty pages. Valid values are 0-255, with a default of zero. • <i>IDENTITY_GAP</i> controls the gap between consecutive values in identity columns. • <i>EXTERNAL TABLE</i> specifies that the table is stored externally. This is the default, so its usage is optional.
CREATE TRIGGER	<p>Sybase offers support similar to SQL Server, with the following exceptions:</p> <ul style="list-style-type: none"> • <i>WITH ENCRYPTION</i> is not permitted. • <i>AFTER</i> and <i>INSTEAD OF</i> are not permitted. • <i>WITH APPEND</i> is not permitted. • <i>NOT FOR REPLICATION</i> is not permitted. • <i>COLUMNS_UPDATED()</i> function is not supported.
CREATE VIEW	<p>Support is identical to that offered by SQL Server except that Sybase has no <i>ENCRYPTION</i>, <i>SCHEMABINDING</i>, or <i>VIEW_METADATA</i> options.</p>
DECLARE CURSOR	<p>Support is identical to that offered by SQL Server, except that Sybase has no <i>INSENSITIVE</i> or <i>SCROLL</i> options.</p>
DELETE	<p>The <i>DELETE</i> statement is nearly identical between Sybase and SQL Server. SQL Server supports a <i>WITH</i> and <i>OPTION</i> clause that Sybase does not. The features unique to Sybase are listed below.</p> <pre>DELETE [[owner.]{table_name view_name}] [FROM {[owner.] {view_name [READPAST] table_name [READPAST] [(INDEX {index_name} table_name} [PREFETCH size][LRU MRU]] }},...] [WHERE {search_conditions CURRENT OF cursor_name}]] [PLAN "abstract_plan"]</pre> <p><i>READPAST</i> instructs the server to skip over all pages or rows currently locked by other transactions, deleting rows only from pages not currently in use.</p> <p><i>PREFETCH</i> specifies the I/O size, in kilobytes, for tables that are bound to caches.</p> <p><i>LRU/MRU</i> specifies either a least or most recently used buffer replacement strategy.</p> <p><i>PLAN</i> sends <i>abstract_plan</i> as an alternative execution plan to the query optimizer.</p>
DISCONNECT	<p>Supported with the following command:</p> <pre>DISCONNECT</pre>
	<p>Supported with the following syntax:</p> <pre>DROP ROLE role_name [WITH OVERRIDE]</pre>

<i>DROP ROLE</i>	Using <i>WITH OVERRIDE</i> will ignore all restrictions on dropping the role from the databases.
<i>FETCH</i>	<p>The <i>FETCH</i> command differs significantly from SQL Server. Sybase has sequential server-side cursors, so the <i>NEXT</i>, <i>PRIOR</i>, <i>FIRST</i>, and <i>LAST</i> options are not permitted.</p> <p>Additionally, parameters can be used in a target list as long as there is a one-to-one mapping between these items and the ones returned by the <i>SELECT</i> statement when the cursor was created.</p> <p><i>FETCH cursor_name [INTO fetch_target_list]</i></p> <p>For example:</p> <pre> DECLARE authors_cursor CURSOR FOR SELECT au_lname, au_fname FROM authors OPEN authors_cursor FETCH authors_cursor INTO @lname, @fname GO </pre>
<i>GRANT</i>	Support is identical to that offered by SQL Server, except that Sybase has no <i>AS</i> clause.
<i>INSERT</i>	Sybase offers the SQL99 syntax for <i>INSERT</i> statements; therefore, Sybase permits none of the extended features of SQL Server.
<i>RETURN</i>	Sybase's return statement <i>RETURN</i> , unlike SQL Server, cannot return NULL values.
<i>REVOKE</i>	<p>Sybase offers support similar to SQL Server with the following exceptions:</p> <ul style="list-style-type: none"> • The <i>TO</i> clause cannot be used in place of <i>FROM</i>. • The <i>AS</i> clause is not permitted.
<i>SAVEPOINT</i>	Support identical to SQL Server's <i>SAVE TRANSACTION</i> statement.
<i>SELECT</i>	<p>Sybase offers support for <i>SELECT</i> statements identical to SQL Server, with the following exceptions:</p> <ul style="list-style-type: none"> • Sybase has a <i>PLAN</i> clause instead of an <i>OPTION</i> clause used for passing hints to the server's statement optimizer. • Sybase has no <i>TOP</i> clause. • Sybase does not support <i>WITH CUBE</i> or <i>WITH ROLLUP</i> in the <i>GROUP BY</i> clause. • Sybase offers a <i>FOR { UPDATE READ ONLY }</i> clause that can only be used with a stored procedure when the query defines the result for a cursor. • Sybase has <i>AT ISOLATION { 0, 1, 2, 3 }</i> for choosing a non-default isolation level for query execution. • Sybase also offers a <i>SELECT INTO</i> statement, but with an additional, optional clause for controlling the locking: <i>LOCK { DATAROWS DATAPAGES ALLPAGES }</i>
<i>SET ROLE</i>	<p>Sybase supports the <i>SET ROLE</i> statement with the following syntax:</p> <pre> SET ROLE {"SA_ROLE" "SSO_ROLE" "OPER_ROLE" role_name [WITH PASSWD "password"]} { ON OFF } </pre>
<i>SET TIME ZONE</i>	The <i>SET TIMEZONE</i> statement is not supported in Sybase or SQL Server.
<i>START TRANSACTION</i>	Support is offered through <i>BEGIN TRANSACTION</i> , which is identical to SQL Server, except that parameterized transaction names are not supported.
	The <i>UPDATE</i> statement is nearly identical between Sybase and SQL Server. SQL Server supports

UPDATE	<p>WITH and OPTION clauses that Sybase does not. The features unique to Sybase are listed below.</p> <pre>UPDATE {table_name view_name} SET {[table_name.view_name.}] column_name1 = {expression1 NULL (select_statement)} variable_name1 = {expression1 NULL (select_statement)} [,...] [FROM {view_name [READPAST]] table_name [READPAST] [(INDEX {index_name table_name } [PREFETCH size][LRU MRU])][,...] [WHERE {search_conditions CURRENT OF cursor_name}] [PLAN "abstract_plan"]</pre>
UPDATE	<ul style="list-style-type: none">• READPAST instructs the server to skip over all pages or rows currently locked by other transactions, updating rows only from pages not currently in use.• PREFETCH specifies the I/O size, in kilobytes, for tables that are bound to caches.• LRU/MRU specifies either the least or most recently used buffer replacement strategy.• PLAN sends abstract_plan as an alternative execution plan to the query optimizer.

A.3 Sybase Adaptive Server SQL Functions

This section compares the functions provided by Sybase and SQL Server. [Table A-3](#) provides a list of the functions support differ on SQL Server. Only functions with differences between the two platforms have been listed.

Table A-3. Sybase Adaptive Server-supported function

Sybase function	SQL Server function	
To retrieve the application name on Sybase, use the following SQL statement: <pre>select program_name from master..sysprocesses where spid = @@spid</pre>	<i>APP_NAME()</i>	Retu by a
N/A	<i>BINARY_CHECKSUM({* expression [,...]})</i>	Retu row
N/A	<i>CAST(expression AS datatype)</i>	Conv spec Ente
<i>CHAR_LENGTH(expression)</i>	N/A	Retu expr prov
N/A	<i>CHECKSUM({* expression [,...]})</i>	Retu valu
N/A	<i>CHECKSUM_AGG([ALL DISTINCT] expression)</i>	Retu
N/A	<i>COALESCE(expression [,...])</i>	Retu argu
<i>COMPARE(char_expression1, char_expression2 [, {collation_name collation_ID}])</i>	N/A	Retu colla 1 : c char. 0: cf -1 : char.
N/A	<i>CONTAINS({column * }, 'contains_search_condition')</i>	Sear cont func to th
N/A	<i>CONTAINSTABLE(table, column, contains_search_condition)</i>	Retu cont func to th
N/A	<i>COUNT_BIG([ALL DISTINCT] expression)</i>	Sam
N/A	<i>DATABASEPROPERTYEX(database, property)</i>	Retu
N/A	<i>DAY(date)</i>	Retu the c
N/A	<i>FILE_ID('file_name')</i>	Retu
N/A	<i>FILE_NAME(file_id)</i>	Retu
N/A	<i>FILEGROUP_ID('filegroup_name')</i>	Retu
N/A	<i>FILEGROUP_NAME(filegroup_id)</i>	Retu
N/A	<i>FILEGROUPPROPERTY(filegroup_name, property)</i>	Retu prop
N/A	<i>FILEPROPERTY(file, property)</i>	Retu prop

N/A	<i>FULLTEXTCATALOGPROPERTY</i> (catalog_name, property)	Retu
N/A	<i>FULLTEXTSERVICEPROPERTY</i> (property)	Retu
N/A	<i>FORMATMESSAGE</i> (msg_number, param_value [,...])	Cons SYS
N/A	<i>FREETEXTTABLE</i> (table { column *}, 'freetext_string' [, top_n_by_rank])	Usec colur matc
N/A	<i>GETANSINULL</i> (['database'])	Retu
N/A	<i>GETUTCDATE</i> ()	Retu
N/A	<i>GROUPING</i> (column_name)	Retu ROLI
<i>HEXTOINT</i> (hexadecimal_string)	N/A	Retu argu CON How hexa with
N/A	<i>IDENT_INCR</i> ('table_or_view')	Retu
N/A	<i>IDENT_SEED</i> ('table_or_view')	Retu
N/A	<i>IDENT_CURRENT</i> ('table_name')	Retu spec
N/A	<i>IDENTITY</i> (data_type [, seed, increment]) As column_name	Usec ideni
N/A	<i>INDEXPROPERTY</i> (table_id, index, property)	Retu
<i>INTTOHEX</i> (integer_expression)	N/A	Retu integ SQL integ
N/A	<i>ISDATE</i> (expression)	Valic DATi
N/A	<i>IS_MEMBER</i> ({'group' 'role'})	Retu whet Serv Syba
N/A	<i>IS_SRVROLEMEMBER</i> ('role' [, 'login'])	Retu whet
N/A	<i>ISNUMERIC</i> (expression)	Valic NUM use i SELE NOT
<i>IS_SEC_SERVICE_ON</i> ('security_service')	N/A	Retu 0 ott
<i>lct_admin</i> ({'LASTCHANCE' 'LOGFULL'})	N/A	Modi proc proc
N/A	<i>LEFT</i> (character_expression, integer_expression)	Retu start
N/A	<i>LEN</i> (string_expression)	Retu expr funci
<i>LICENSE_ENABLED</i> ('feature')	N/A	Retu
N/A.	<i>MONTH</i> (date)	Retu
<i>MUT_EXCL_ROLES</i> (role1, role2 [, membership [, activation]])	N/A	Retu rovl

<i>{membership activation }</i>		Execu
N/A	<i>NCHAR(integer_expression)</i>	Retu integ funci
N/A	<i>NEWID()</i>	Crea UNIC
N/A	<i>NULLIF(expression, expression)</i>	Retu equi
N/A	<i>OBJECTPROPERTY(id, property)</i>	Retu data
N/A	<i>OPEN{[Global]cursor_name cursor_variable_name}</i>	Oper
N/A	<i>OPENDATASOURCE(provider_name, init_string)</i>	Mak linke
N/A	<i>OPENQUERY(linked_server, 'query')</i>	Quer a lin
N/A	<i>OPENROWSET('provider_name', {'datasource'; 'user_id',password 'provider_string'},{[catalog.][schema.]object 'query'})</i>	Quer as a
N/A	<i>PARSENAME('object_name', object_piece)</i>	Retu nam <i>Obje</i>
<i>PATINDEX('%pattern%', expression)</i>	N/A	Retu patt
N/A	<i>PERMISSIONS(object_id, column)</i>	Retu user
<i>PROC_ROLE('role_name')</i>	N/A	Retu the r the j funci
N/A	<i>REPLACE('string_expression1','string_expression2','string_expression3')</i>	Repl with Sam
<i>ROLE_CONTAIN('role1','role2')</i>	N/A	Retu
<i>ROLE_ID(role_name)</i>	N/A	Retu <i>role_</i> <i>state</i> <i>SELE</i> <i>issqlr</i> This role
<i>ROLE_NAME(role_id)</i>	N/A	Retu equ state <i>SELE</i> <i>uid =</i>
<i>ROWCNT(sysindexes.doampg)</i>	N/A	Prov table use f <i>SELE</i> <i>WHEI</i>
N/A	<i>ROWCOUNT_BIG()</i>	Retu (sam

N/A	<i>SESSION_USER</i>	Retu
<i>SHOW_ROLE()</i>	N/A	Retu
<i>SHOW_SEC_SERVICES()</i>	N/A	Retu
<i>SORTKEY(char_expr [, {collation_name collation_id}])</i>	N/A	Retu valu
N/A	<i>STATS_DATE(table_id, index_id)</i>	Retu last
N/A	<i>STDEV(expression)</i>	Retu colur
N/A	<i>STDEVP(expression)</i>	Retu valu
<i>SUSER_ID(['login'])</i>	N/A	Retu spec func
N/A	<i>SUSER_SID(['login'])</i>	Retu spec equi
<i>SUSER_NAME([server_user_id])</i>	N/A	Retu spec equi
N/A	<i>SUSER_SNAME([server_user_sid])</i>	Retu spec equi
<i>SYB_SENDMSG(ip_address, port, message)</i>	N/A	Send argu
N/A	<i>SYSTEM_USER</i>	Retu
<i>TO_UNICHAR(integer)</i>	N/A	Retu equi use 1
<i>TSEQUEL(timestamp_expr, timestamp_literal)</i>	N/A	Retu time Serv equi
N/A	<i>TYPEPROPERTY(datatype,property)</i>	Retu
<i>UHIGHSURR(uchar_expr, start)</i>	N/A	Retu posit
<i>ULOWSURR(uchar_expr, start)</i>	N/A	Retu posit
N/A	<i>UNICODE('ncharacter_expression')</i>	Retu char equi
<i>USCALAR(uchar_expr)</i>	N/A	Retu char equi
<i>VALID_NAME(user_name)</i>	N/A	Retu user
<i>VALID_USER(user_id)</i>	N/A	Retu iden
N/A	<i>VAR(expression)</i>	Retu
N/A	<i>VARP(expression)</i>	Retu valu
N/A	<i>YEAR(date)</i>	Retu spec

A.4 Sybase Adaptive Server Keywords

This section lists the keywords used by Sybase Adaptive Server Enterprise.

ABSOLUTE	ACTION	ADD	ALL
ALLOCATE	ALTER	AND	ANY
ARE	ARITH_OVERFLOW	AS	ASC
ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH
BOTH	BREAK	BROWSE	BULK
BY	CASCADE	CASCADED	CASE
CAST	CATALOG	CHAR	CHAR_CONVERT
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CHECKPOINT	CLOSE	CLUSTERED	COALESCE
COLLATE	COLLATION	COLUMN	COMMIT
COMPUTE	CONFIRM	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONTINUE	CONTROLROW
CONVERT	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DATE	DAY	DBCC	DEALLOCATE
DEC	DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE	DESC
DESCRIBE	DESCRIPTOR	DETERMINISTIC	DIAGNOSTICS
DISCONNECT	DISK DISTINCT	DOMAIN	DOUBLE
DROP	DUMMY	DUMP	ELSE
END	END-EXEC	ENDTRAN	ERRLVL
ERRORDATA	ERROREXIT	ESCAPE	EXCEPT
EXCEPTION	EXCLUSIVE	EXEC	EXECUTE
EXISTS	EXIT	EXP_ROW_SIZE	EXTERNAL
EXTRACT	FALSE	FETCH	FILLFACTOR
FIRST	FLOAT	FOR	FOREIGN
FOUND	FROM	FULL	FUNC
FUNCTION	GET	GLOBAL	GO
GOTO	GRANT	GROUP	HAVING
HOLDLOCK	HOUR	IDENTITY	IDENTITY_GAP
IDENTITY_INSERT	IDENTITY_START	IF	IMMEDIATE
IN	INDEX	INDICATOR	INITIALLY
INNER	INOUT	INPUT	INSENSITIVE
INSERT	INSTALL	INT	INTEGER
INTERSECT	INTERVAL	INTO	IS
ISOLATION	JAR	JOIN	KEY
KILL	LANGUAGE	LAST	LEADING

LEFT	LEVEL	LIKE	LINENO
LOAD	LOCAL	LOCK	LOWER
MATCH	MAX	MAX_ROWS_PER_PAGE	MIN
MINUTE	MIRROR	MIRROREXIT	MODIFY
MODULE	MONTH	NAMES	NATIONAL
NATURAL	NCHAR	NEW	NEXT
NO	NOHOLDLOCK	NONCLUSTERED	NOT
NULL	NULLIF	NUMERIC	NUMERIC_TRUNCATION
OCTET_LENGTH	OF	OFF	OFFSETS
ON	ONCE	ONLINE	ONLY
OPEN	OPTION	OR	ORDER
OUT	OUTER	OUTPUT	OVER
OVERLAPS	PAD	PARTIAL	PARTITION
PERM	PERMANENT	PLAN	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY
PRINT	PRIOR	PRIVILEGES	PROC
PROCEDURE	PROCESSEXIT	PROXY_TABLE	PUBLIC
QUIESCE	RAISERROR	READ	READPAST
READTEXT	REAL	RECONFIGURE	REFERENCES REMOVE
RELATIVE	REORG	REPLACE	REPLICATION
RESERVEPAGEGAP	RESTRICT	RETURN	RETURNS
REVOKE	RIGHT	ROLE	ROLLBACK
ROWCOUNT	ROWS	RULE	SAVE
SCHEMA	SCROLL	SECOND	SECTION
SELECT	SESSION_USER	SET	SETUSER
SHARED	SHUTDOWN	SIZE	SMALLINT
SOME	SPACE	SQL	SQLCODE
SQLERROR	SQLSTATE	STATISTICS	STRINGSIZE
STRIPE	SUBSTRING	SUM	SYB_IDENTITY
SYB_RESTREE	SYB_TERMINATE	SYSTEM_USER	TABLE
TEMP	TEMPORARY	TEXTSIZE	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRAN	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER	TRIM
TRUE	TRUNCATE	TSEQUAL	UNION
UNIQUE	UNKNOWN	UNPARTITION	UPDATE
UPPER	USAGE	USE	USER
USER_OPTION	USING	VALUE	VALUES
VARCHAR	VARYING	VIEW	WAITFOR
WHEN	WHENEVER	WHERE	WHILE
WITH	WORK	WRITE	WRITETEXT
YEAR	ZONE		





Appendix A. Sybase Adaptive Server

For the most part, Microsoft SQL Server's dialect of SQL documented throughout this book applies to Sybase Adaptive Server Enterprise 12.5. This appendix documents those datatypes, statements, functions, and keywords that differ between these two popular platforms.



Appendix B. Shared and Platform-Specific Keywords

The tables below display the keywords in the ANSI standard, and in the five platform-specific implementations of SQL that are discussed in this book, with the Sybase Adaptive Server. The keyword tables appear in the following order:

- [Shared keywords](#)
- [SQL2003 keywords](#)
- [DB2 keywords](#)
- [MySQL keywords](#)
- [Oracle keywords](#)
- [PostgreSQL keywords](#)
- [SQL Server keywords](#)
- [Sybase Adaptive Server keywords](#)

Table B-1. Shared keywords

ADD	ALL	ALTER	AND
AS	ASC	BY	CHECK
COLUMN	CREATE	DATE	DEFAULT
DELETE	DESC	DROP	FOR
FROM	IN	INTO	IS
LIKE	NOT	NULL	ON
OR	ORDER	REVOKE	SELECT
SET	TABLE	THEN	TO
UNIQUE	UPDATE	WITH	

Table B-2. SQL2003 keywords

ABSOLUTE	ACTION	ADD	ADMIN
AFTER	AGGREGATE	ALIAS	ALL
ALLOCATE	ALTER	AND	ANY
ARE	ARRAY	AS	ASC
ASSERTION	ASSERTION	AT	ATOMIC
AUTHORIZATION	BEFORE	BEGIN	BIGINT
BINARY	BIT	BLOB	BOOLEAN
BOTH	BREADTH	BY	CALL
CASCADE	CASCADEDE	CASE	CAST
CATALOG	CHAR	CHARACTER	CHECK
CLASS	CLOB	CLOSE	COLLATE
COLLATION	COLLECT	COLUMN	COMMIT
COMPLETION	CONDITION	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONSTRUCTOR	CONTAINS

CONTINUE	CORRESPONDING	CREATE	CROSS
CUBE	CURRENT	CURRENT_DATE	CURRENT_PATH
CURRENT_ROLE	CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER
CURSOR	CYCLE	DATA	DATALINK
DATE	DAY	DEALLOCATE	DEC
DECIMAL	DECLARE	DEFAULT	DEFERRABLE
DELETE	DEPTH	DEREF	DESC
DESCRIPTOR	DESTRUCTOR	DIAGNOSTICS	DICTIONARY
DISCONNECT	DO	DOMAIN	DOUBLE
DROP	ELEMENT	END-EXEC	EQUALS
ESCAPE	EXCEPT	EXCEPTION	EXECUTE
EXIT	EXPAND	EXPANDING	FALSE
FIRST	FLOAT	FOR	FOREIGN
FREE	FROM	FUNCTION	FUSION
GENERAL	GET	GLOBAL	GOTO
GROUP	GROUPING	HANDLER	HASH
HOUR	IDENTITY	IF	IGNORE
IMMEDIATE	IN	INDICATOR	INITIALIZE
INITIALLY	INNER	INOUT	INPUT
INSERT	INT	INTEGER	INTERSECT
INTERSECTION	INTERVAL	INTO	IS
ISOLATION	ITERATE	JOIN	KEY
LANGUAGE	LARGE	LAST	LATERAL
LEADING	LEAVE	LEFT	LESS
LEVEL	LIKE	LIMIT	LOCAL
LOCALTIME	LOCALTIMESTAMP	LOCATOR	LOOP
MATCH	MEMBER	MEETS	MERGE
MINUTE	MODIFIES	MODIFY	MODULE
MONTH	MULTISET	NAMES	NATIONAL
NATURAL	NCHAR	NCLOB	NEW
NEXT	NO	NONE	NORMALIZE
NOT	NULL	NUMERIC	OBJECT
OF	OFF	OLD	ON
ONLY	OPEN	OPERATION	OPTION
OR	ORDER	ORDINALITY	OUT
OUTER	OUTPUT	PAD	PARAMETER
PARAMETERS	PARTIAL	PATH	PERIOD
POSTFIX	PRECEDES	PRECISION	PREFIX
PREORDER	PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVILEGES	PROCEDURE	PUBLIC
READ	READS	REAL	RECURSIVE
REDO	REF	REFERENCES	REFERENCING
RELATIVE	REPEAT	RESIGNAL	RESTRICT
RESULT	RETURN	RETURNS	REVOKE

RIGHT	ROLE	ROLLBACK	ROLLUP
ROUTINE	ROW	ROWS	SAVEPOINT
SCHEMA	SCROLL	SEARCH	SECOND
SECTION	SELECT	SEQUENCE	SESSION
SESSION_USER	SET	SETS	SIGNAL
SIZE	SMALLINT	SPECIFIC	SPECIFICTYPE
SQL	SQLEXCEPTION	SQLSTATE	SQLWARNING
START	STATE	STATIC	STRUCTURE
SUBMULTISET	SUCCEEDS	SUM	SYSTEM_USER
TABLE	TABLESAMPLE	TEMPORARY	TERMINATE
THAN	THEN	TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE	TO	TRAILING
TRANSACTION	TRANSLATION	TREAT	TRIGGER
TRUE	UESCAPE	UNDER	UNDO
UNION	UNIQUE	UNKNOWN	UNTIL
UPDATE	USAGE	USER	USING
VALUE	VALUES	VARCHAR	VARIABLE
VARYING	VIEW	WHEN	WHENEVER
WHERE	WHILE	WITH	WRITE
YEAR	ZONE		

Table B-3. DB2 keywords

ACQUIRE	ADD	AFTER	ALIAS
ALL	ALLOCATE	ALLOW	ALTER
AND	ANY	AS	ASC
ASUTIME	AUDIT	AUTHORIZATION	AUXILIARY
AVG	BEFORE	BEGIN	BETWEEN
BINARY	BUFFERPOOL	BY	CALL
CALLED	CAPTURE	CASCADE	CASE
CAST	CCSID	CHAR	CHARACTER
CHECK	CLOSE	CLUSTER	COLLECTION
COLLID	COLUMN	COMMENT	COMMIT
CONCAT	CONDITION	CONNECT	CONNECTION
CONSTRAINT	CONTAINS	CONTINUE	COUNT
COUNT_BIG	CREATE	CROSS	CURRENT
CURRENT_DATE	CURRENT_LC_PATH	CURRENT_PATH	CURRENT_SERVER
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_TIMEZONE	CURRENT_USER
CURSOR	DATA	DATABASE	DATE
DAY	DAYS	DB2GENERAL	DB2SQL
DBA	DBINFO	DBSPACE	DECLARE
DEFAULT	DELETE	DESC	DESCRIPTOR
DETERMINISTIC	DISALLOW	DISCONNECT	DISTINCT
DO	DOUBLE	DROP	DSSIZE

DYNAMIC	EDITPROC	ELSE	ELSEIF
END	END-EXEC	ERASE	ESCAPE
EXCEPT	EXCEPTION	EXCLUSIVE	EXECUTE
EXISTS	EXIT	EXPLAIN	EXTERNAL
FENCED	FETCH	FIELDPROC	FILE
FINAL	FOR	FOREIGN	FREE
FROM	FULL	FUNCTION	GENERAL
GENERATED	GO	GOTO	GRANT
GRAPHIC	GROUP	HANDLER	HAVING
HOUR	HOURS	IDENTIFIED	IF
IMMEDIATE	IN	INDEX	INDICATOR
INNER	INOUT	INSENSITIVE	INSERT
INTEGRITY	INTERSECT	INTO	IS
ISOBID	ISOLATION	JAVA	JOIN
KEY	LABEL	LANGUAGE	LC_CTYPE
LEAVE	LEFT	LIKE	LINKTYPE
LOCAL	LOCALE	LOCATOR	LOCATORS
LOCK	LOCKSIZE	LONG	LOOP
MAX	MICROSECOND	MICROSECONDS	MIN
MINUTE	MINUTES	MODE	MODIFIES
MONTH	MONTHS	NAME	NAMED
NHEADER	NO	NODENAME	NODENUMBER
NOT	NULL	NULLS	NUMPARTS
OBID	OF	ON	ONLY
OPEN	OPTIMIZATION	OPTIMIZE	OPTION
OR	ORDER	OUT	OUTER
PACKAGE	PAGE	PAGES	PARAMETER
PART	PARTITION	PATH	PCTFREE
PCTINDEX	PIECESIZE	PLAN	POSITION
PRECISION	PREPARE	PRIMARY	PRIQTY
PRIVATE	PRIVILEGES	PROCEDURE	PROGRAM
PSID	PUBLIC	QUERYNO	READ
READS	RECOVERY	REFERENCES	RELEASE
RENAME	REPEAT	RESET	RESOURCE
RESTRICT	RESULT	RETURN	RETURNS
REVOKE	RIGHT	ROLLBACK	ROW
ROWS	RRN	RUN	SCHEDULE
SCHEMA	SCRATCHPAD	SECOND	SECONDS
SECQTY	SECURITY	SELECT	SET
SHARE	SIMPLE	SOME	SOURCE
SPECIFIC	SQL	STANDARD	STATIC
STATISTICS	STAY	STOGROUP	STORES
STORPOOL	STYLE	SUBPAGES	SUBSTRING
SUM	SYNONYM	TABLE	TABLESPACE
THEN	TO	TRANSACTION	TRIGGER

TRIM	TYPE	UNDO	UNION
UNIQUE	UNTIL	UPDATE	USAGE
USER	USING	VALIDPROC	VALUES
VARIABLE	VARIANT	VCAT	VIEW
VOLUMES	WHEN	WHERE	WHILE
WITH	WLM	WORK	WRITE
YEAR	YEARS		

Table B-4. MySQL keywords

ACTION	ADD	AFTER	AGGREGATE
ALL	ALTER	AND	AS
ASC	AUTO_INCREMENT	AVG	AVG_ROW_LENGTH
BETWEEN	BIGINT	BINARY	BIT
BLOB	BOOL	BOTH	BY
CASCADE	CASE	CHANGE	CHAR
CHARACTER	CHECK	CHECKSUM	COLUMN
COLUMNS	COMMENT	CONSTRAINT	CREATE
CROSS	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
DATA	DATABASE	DATABASES	DATE
DATETIME	DAY	DAY_HOUR	DAY_MINUTE
DAY_SECOND	DAYOFMONTH	DAYOFWEEK	DAYOFYEAR
DEC	DECIMAL	DEFAULT	DELAY_KEY_WRITE
DELAYED	DELETE	DESC	DESCRIBE
DISTINCT	DISTINCTROW	DOUBLE	DROP
ELSE	ENCLOSED	END	ENUM
ESCAPE	ESCAPED	EXISTS	EXPLAIN
FIELDS	FILE	FIRST	FLOAT
FLOAT4	FLOAT8	FLUSH	FOR
FOREIGN	FROM	FULL	FUNCTION
GLOBAL	GRANT	GRANTS	GROUP
HAVING	HEAP	HIGH_PRIORITY	HOSTS
HOURL	HOURL_MINUTE	HOURL_SECOND	IDENTIFIED
IF	IGNORE	IN	INDEX
INFILE	INNER	INSERT	INSERT_ID
INT	INT1	INT2	INT3
INT4	INT8	INTEGER	INTERVAL
INTO	IS	ISAM	JOIN
KEY	KEYS	KILL	LAST_INSERT_ID
LEADING	LEFT	LENGTH	LIKE
LIMIT	LINES	LOAD	LOCAL
LOCK	LOGS	LONG	LOB
LONGTEXT	LOW_PRIORITY	MATCH	MAX
MAX_ROWS	MEDIUMLOB	MEDIUMINT	MEDIUMTEXT

MIDDLEINT	MIN_ROWS	MINUTE	MINUTE_SECOND
MODIFY	MONTH	MONTHNAME	MYISAM
NATURAL	NO	NOT	NULL
NUMERIC	ON	OPTIMIZE	OPTION
OPTIONALLY	OR	ORDER	OUTER
OUTFILE	PACK_KEYS	PARTIAL	PASSWORD
PRECISION	PRIMARY	PRIVILEGES	PROCEDURE
PROCESS	PROCESLIST	READ	REAL
REFERENCES	REGEXP	RELOAD	RENAME
REPLACE	RESTRICT	RETURNS	REVOKE
RLIKE	ROW	ROWS	SECOND
SELECT	SET	SHOW	SHUTDOWN
SMALLINT	SONAME	SQL_BIG_RESULT	SQL_BIG_SELECTS
SQL_BIG_TABLES	SQL_LOG_OFF	SQL_LOG_UPDATE	SQL_LOW_PRIORITY_UPDATES
SQL_SELECT_LIMIT	SQL_SMALL_RESULT	SQL_WARNINGS	STARTING
STATUS	STRAIGHT_JOIN	STRING	TABLE
TABLES	TEMPORARY	TERMINATED	TEXT
THEN	TIME	TIMESTAMP	TINYBLOB
TINYINT	TINYTEXT	TO	TRAILING
TYPE	UNIQUE	UNLOCK	UNSIGNED
UPDATE	USAGE	USE	USING
VALUES	VARBINARY	VARCHAR	VARIABLES
VARYING	WHEN	WITH	WRITE
ZEROFILL			

Table B-5. Oracle keywords

ACCESS	ADD	ALL	ALTER
AND	ANY	AS	ASC
AUDIT	BETWEEN	BY	CHAR
CHECK	CLUSTER	COLUMN	COMMENT
COMPRESS	CONNECT	CREATE	CURRENT
DATE	DECIMAL	DEFAULT	DELETE
DESC	DISTINCT	DROP	ELSE
EXCLUSIVE	EXISTS	FILE	FLOAT
FOR	FROM	GRANT	GROUP
HAVING	IDENTIFIED	IMMEDIATE	IN
INCREMENT	INDEX	INITIAL	INSERT
INTEGER	INTERSECT	INTO	IS
LEVEL	LIKE	LOCK	LONG
MAXEXTENTS	MINUS	MODE	MODIFY
MSLABEL	NOAUDIT	NOCOMPRESS	NOT
NOWAIT	NULL	NUMBER	OF
OFFLINE	ON	ONLINE	OPTION

OR	ORDER	PCTFREE	PRIOR
PRIVILEGES	PUBLIC	RAW	RENAME
RESOURCE	REVOKE	ROW	ROWID
ROWNUM	ROWS	SELECT	SESSION
SET	SHARE	SIZE	SMALLINT
START	SUCCESSFUL	SYNONYM	SYSDATE
TABLE	THEN	TO	TRIGGER
UID	UNION	UNIQUE	UPDATE
USER	VALIDATE	VALUES	VARCHAR
VARCHAR2	VIEW	WHENEVER	WHERE
WITH			

Table B-6. PostgreSQL keywords

ABORT	ADD	ALL	ALLOCATE
ALTER	ANALYZE	AND	ANY
ARE	AS	ASC	ASSERTION
AT	AUTHORIZATION	AVG	BEGIN
BETWEEN	BINARY	BIT	BIT_LENGTH
BOTH	BY	CASCADE	CASCADEDE
CASE	CAST	CATALOG	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CLOSE	CLUSTER	COALESCE	COLLATE
COLLATION	COLUMN	COMMIT	CONNECT
CONNECTION	CONSTRAINT	CONTINUE	CONVERT
COPY	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_SESSION
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATE	DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DELETE	DESC
DESCRIBE	DESCRIPTOR	DIAGNOSTICS	DISCONNECT
DISTINCT	DO	DOMAIN	DROP
ELSE	END	ESCAPE	EXCEPT
EXCEPTION	EXEC	EXECUTE	EXISTS
EXISTS	EXPLAIN	EXTEND	EXTERNAL
EXTRACT	FALSE	FETCH	FIRST
FLOAT	FOR	FOREIGN	FOUND
FROM	FULL	FULL	GET
GLOBAL	GO	GOTO	GRANT
GROUP	HAVING	IDENTITY	IN
INDICATOR	INNER	INPUT	INSERT
INTERSECT	INTERVAL	INTO	IS
JOIN	LAST	LEADING	LEFT
LIKE	LISTEN	LOAD	LOCAL

LOCK	LOWER	MAX	MIN
MODULE	MOVE	NAMES	NATIONAL
NATURAL	NCHAR	NEW	NO
NONE	NOT	NOTIFY	NULL
NULLIF	NUMERIC	OCTET_LENGTH	OFFSET
ON	OPEN	OR	ORDER
OUTER	OUTPUT	OVERLAPS	PARTIAL
POSITION	PRECISION	PREPARE	PRESERVE
PRIMARY	PRIVILEGES	PROCEDURE	PUBLIC
REFERENCES	RESET	REVOKE	RIGHT
ROLLBACK	ROWS	SCHEMA	SECTION
SELECT	SESSION	SESSION_USER	SET
SETOF	SHOW	SIZE	SOME
SQL	SQLCODE	SQLERROR	SQLSTATE
SUBSTRING	SUM	SYSTEM_USER	TABLE
TEMPORARY	THEN	TO	TO
TRAILING	TRANSACTION	TRANSLATE	TRANSLATION
TRIM	TRUE	UNION	UNIQUE
UNIQUE	UNKNOWN	UNLISTEN	UNTIL
UPDATE	UPPER	USAGE	USER
USING	VACUUM	VALUE	VALUES
VARCHAR	VARYING	VERBOSE	VIEW
WHEN	WHENEVER	WHERE	WITH
WORK	WRITE		

Table B-7. Microsoft SQL Server keywords

ADD	ALL	ALTER	AND
ANY	AS	ASC	AUTHORIZATION
BACKUP	BEGIN	BETWEEN	BREAK
BROWSE	BULK	BY	CASCADE
CASE	CHECK	CHECKPOINT	CLOSE
CLUSTERED	COALESCE	COLLATE	COLUMN
COMMIT	COMPUTE	CONSTRAINT	CONTAINS
CONTAINSTABLE	CONTINUE	CONVERT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DBCC	DEALLOCATE	DECLARE	DEFAULT
DELETE	DENY	DESC	DISK
DISTINCT	DISTRIBUTED	DOUBLE	DROP
DUMMY	DUMP	ELSE	END
ERRLVL	EXCEPT	EXEC	EXECUTE
EXISTS	EXIT	FETCH	FILE
FILLFACTOR	FOR	FOREIGN	FREETEXT

FREETEXTTABLE	FROM	FULL	FUNCTION
GOTO	GRANT	GROUP	HAVING
HOLDLOCK	IDENTITY	IDENTITY_INSERT	IDENTITYCOL
IF	IN	INDEX	INNER
INSERT	INTERSECT	INTO	IS
JOIN	KEY	KILL	LEFT
LIKE	LINENO	LOAD	NATIONAL
NOCHECK	NONCLUSTERED	NOT	NULL
NULLIF	OF	OFF	OFFSETS
ON	OPEN	OPENDATASOURCE	OPENQUERY
OPENROWSET	OPENXML	OPTION	OR
ORDER	OUTER	PERCENT	PLAN
PRECISION	PRIMARY	PRINT	PROC
PROCEDURE	PUBLIC	RAISERROR	READ
READTEXT	RECONFIGURE	REFERENCES	REPLICATION
RESTORE	RESTRICT	RETURN	REVOKE
RIGHT	ROLLBACK	ROWCOUNT	ROWGUIDCOL
RULE	SAVE	SCHEMA	SELECT
SESSION_USER	SET	SETUSER	SHUTDOWN
SOME	STATISTICS	SYSTEM_USER	TABLE
TEXTSIZE	THEN	TO	TOP
TRAN	TRANSACTION	TRIGGER	TRUNCATE
TSEQUAL	UNION	UNIQUE	UPDATE
UPDATETEXT	USE	USER	VALUES
VARYING	VIEW	WAITFOR	WHEN
WHERE	WHILE	WITH	WRITETEXT

Table B-8. Sybase Adaptive Server keywords

ABSOLUTE	ACTION	ADD	ALL
ALLOCATE	ALTER	AND	ANY
ARE	ARITH_OVERFLOW	AS	ASC
ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH
BOTH	BREAK	BROWSE	BULK
BY	CASCADE	CASCADED	CASE
CAST	CATALOG	CHAR	CHAR_CONVERT
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CHECKPOINT	CLOSE	CLUSTERED	COALESCE
COLLATE	COLLATION	COLUMN	COMMIT
COMPUTE	CONFIRM	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONTINUE	CONTROLROW
CONVERT	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME

CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DATE	DAY	DBCC	DEALLOCATE
DEC	DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE	DESC
DESCRIBE	DESCRIPTOR	DETERMINISTIC	DIAGNOSTICS
DISCONNECT	DISK DISTINCT	DOMAIN	DOUBLE
DROP	DUMMY	DUMP	ELSE
END	END-EXEC	ENDTRAN	ERRLVL
ERRORDATA	ERROREXIT	ESCAPE	EXCEPT
EXCEPTION	EXCLUSIVE	EXEC	EXECUTE
EXISTS	EXIT	EXP_ROW_SIZE	EXTERNAL
EXTRACT	FALSE	FETCH	FILLFACTOR
FIRST	FLOAT	FOR	FOREIGN
FOUND	FROM	FULL	FUNC
FUNCTION	GET	GLOBAL	GO
GOTO	GRANT	GROUP	HAVING
HOLDLOCK	HOUR	IDENTITY	IDENTITY_GAP
IDENTITY_INSERT	IDENTITY_START	IF	IMMEDIATE
IN	INDEX	INDICATOR	INITIALLY
INNER	INOUT	INPUT	INSENSITIVE
INSERT	INSTALL	INT	INTEGER
INTERSECT	INTERVAL	INTO	IS
ISOLATION	JAR	JOIN	KEY
KILL	LANGUAGE	LAST	LEADING
LEFT	LEVEL	LIKE	LINENO
LOAD	LOCAL	LOCK	LOWER
MATCH	MAX	MAX_ROWS_PER_PAGE	MIN
MINUTE	MIRROR	MIRROREXIT	MODIFY
MODULE	MONTH	NAMES	NATIONAL
NATURAL	NCHAR	NEW	NEXT
NO	NOHOLDLOCK	NONCLUSTERED	NOT
NULL	NULLIF	NUMERIC	NUMERIC_TRUNCATION
OCTET_LENGTH	OF	OFF	OFFSETS
ON	ONCE	ONLINE	ONLY
OPEN	OPTION	OR	ORDER
OUT	OUTER	OUTPUT	OVER
OVERLAPS	PAD	PARTIAL	PARTITION
PERM	PERMANENT	PLAN	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY
PRINT	PRIOR	PRIVILEGES	PROC
PROCEDURE	PROCESSEXIT	PROXY_TABLE	PUBLIC
QUIESCE	RAISERROR	READ	READPAST
READTEXT	REAL	RECONFIGURE	REFERENCES REMOVE
RELATIVE	REORG	REPLACE	REPLICATION
RESERVEPAGEGAP	RESTRICT	RETURN	RETURNS

REVOKE	RIGHT	ROLE	ROLLBACK
ROWCOUNT	ROWS	RULE	SAVE
SCHEMA	SCROLL	SECOND	SECTION
SELECT	SESSION_USER	SET	SETUSER
SHARED	SHUTDOWN	SIZE	SMALLINT
SOME	SPACE	SQL	SQLCODE
SQLERROR	SQLSTATE	STATISTICS	STRINGSIZE
STRIPE	SUBSTRING	SUM	SYB_IDENTITY
SYB_RESTREE	SYB_TERMINATE	SYSTEM_USER	TABLE
TEMP	TEMPORARY	TEXTSIZE	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRAN	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER	TRIM
TRUE	TRUNCATE	TSEQUAL	UNION
UNIQUE	UNKNOWN	UNPARTITION	UPDATE
UPPER	USAGE	USE	USER
USER_OPTION	USING	VALUE	VALUES
VARCHAR	VARYING	VIEW	WAITFOR
WHEN	WHENEVER	WHERE	WHILE
WITH	WORK	WRITE	WRITETEXT
YEAR	ZONE		

1.1 The Relational Model and ANSI SQL

Relational Database Management Systems (RDBMSs), such as those covered in this book, are the primary engines of information systems worldwide, particularly web applications and distributed client/server computing systems. They enable a multitude of users to quickly and simultaneously access, create, edit, and manipulate data without impeding or impacting other users. They also allow developers to write useful applications to access their resources as well as provide administrators with the capabilities they need to maintain, secure, and optimize organizational data resources.

An RDBMS is defined as a system whose users view data as a collection of tables related to each other through common data values. Data is stored in *tables*, which are composed of *rows* and *columns*. Tables of independent data can be linked (or *related*) to one another if they each have unique, identifying columns of data (called *keys*) that represent a data value held in common. E. F. Codd first described relational database theory in his landmark paper "A Relational Model of Data for Large Shared Data Banks," published in the *Communications of the ACM* (Association for Computing Machinery) in June, 1970. Under Codd's new relational data model, data was structured (into tables of rows and columns); manageable using operations like selections, projections, and joins; and consistent as the result of integrity rules like keys and referential integrity. Codd also articulated rules that governed how a relational database should be designed; the process for applying these rules is now known as *normalization*.

1.1.1 Codd's Rules for Relational Database Systems

Codd applied rigorous mathematical theories, primarily set theory, to the management of data, and he compiled a list of criteria a database must meet to be considered relational. At its core, the relational database concept centers around storing data in tables. This concept is now so common as to seem trivial; however, not long ago designing a system capable of sustaining the relational model was considered a long shot with limited usefulness. Following are Codd's *Twelve Principles of Relational Databases*:

1. Information is represented logically in tables.
2. Data must be logically accessible by table, primary key, and column.
3. Null values must be uniformly treated as "missing information," not as empty strings, blanks, or zeros.
4. Metadata (data about the database) must be stored in the database just as regular data is.
5. A single language must be able to define data, views, integrity constraints, authorization, transactions, and data manipulation.
6. Views must show the updates of their base tables and vice versa.
7. A single operation must be available to do each of the following operations: retrieve data, insert data, update data, or delete data.
8. Batch and end-user operations are logically separate from physical storage and access methods.
9. Batch and end-user operations can change the database schema without having to recreate it or the applications built upon it.
10. Integrity constraints must be available and stored in the metadata, not in an application program.
11. The data manipulation language of the relational system should not care where or how the physical data is distributed and should not require alteration if the physical data is centralized or distributed.
12. Any row processing done in the system must obey the same integrity rules and constraints that set-processing operations do.

These principles continue to be the litmus test used to validate the "relational" characteristics of a database platform; a database that does not meet all of these rules is not fully relational. While these rules do not apply to applications development, they do determine whether the database engine itself can be considered truly "relational." Currently, most commercial RDBMS products pass Codd's test. Among the platforms discussed in *SQL in a Nutshell, Second Edition*, only MySQL fails to support all of these requirements.

Understanding Codd's principles assists programmers and developers in the proper development and design of Relational Databases (RDBs). The following sections detail how some of these requirements are met within SQL using RDBs.

1.1.1.1 Data structures (rules 1, 2, and 8)

Codd's rules 1 and 2 state that "information is represented logically in tables" and that "data must be logically

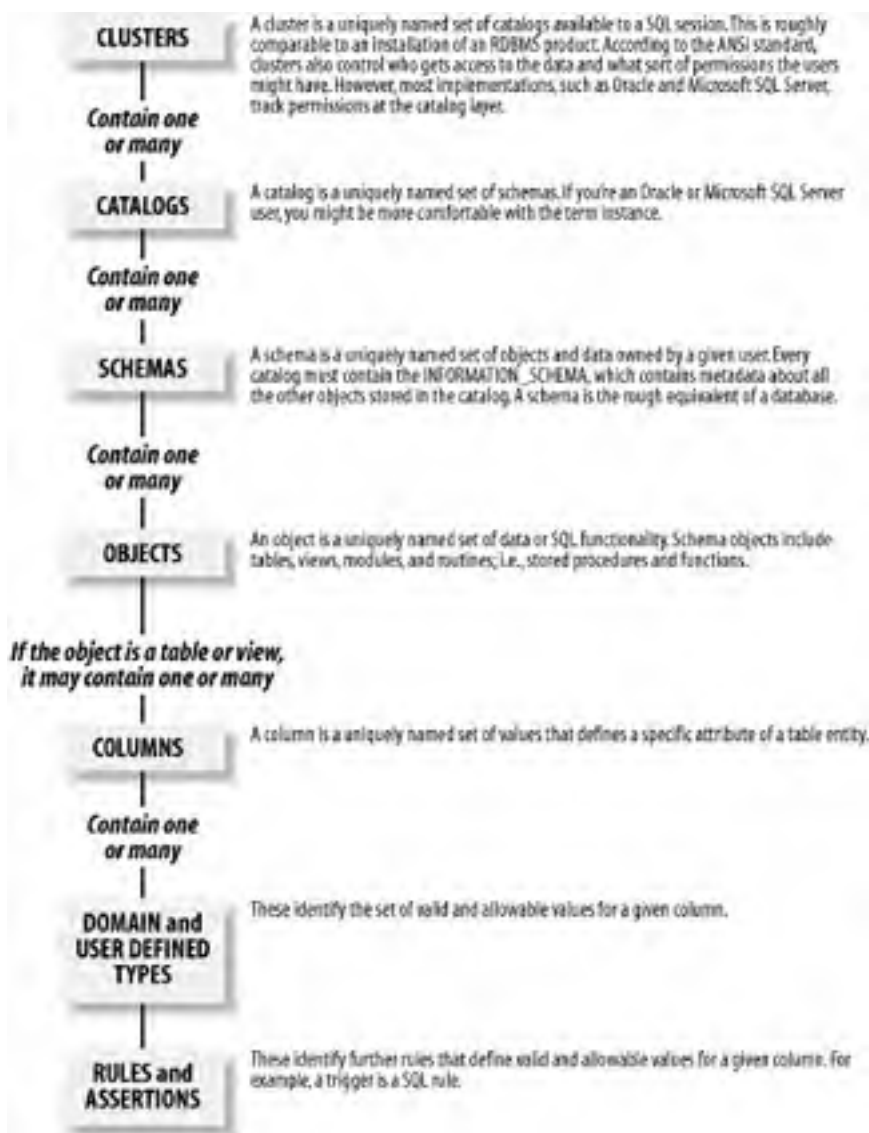
accessible by table, primary key, and column." So the process of defining a table for a SQL database does not require that programs instruct the database how to interact with the underlying physical data structures. Furthermore, SQL logically isolates the process of accessing data and physically maintaining that data as required by rule 8, "batch and end-user operations are logically separate from physical storage and access methods."

In the relational model, data is shown logically as a two-dimensional *table* that describes a single entity (for example, business expenses). Academics refer to tables as *entities* and to columns as *attributes*. Tables are composed of *rows*, or *records* (academics call them *tuples*), and *columns* (called *attributes*, since each column of a table describes a specific attribute of the entity). The intersection of a record and a column provides a single *value*. The column or columns whose values uniquely identify each record can act as a *primary key*. These days this representation seems elementary, but it was actually quite innovative when it was first proposed.

SQL2003 defines a whole data structure hierarchy beyond simple tables, though tables are the core data structure. Relational design handles data on a table-by-table basis, not on a record-by-record basis. This table-centric orientation is the heart of set programming. Consequently, almost all SQL commands operate much more efficiently against sets of data within or across tables than against individual records. Said another way, effective SQL programming requires that you think in terms of sets of data, rather than of individual rows.

Figure 1-1 is a description of the SQL2003 terminology used to describe the hierarchical data structures used by a relational database: *clusters* contain sets of *catalogs*; *catalogs* contain sets of *schemas*; *schemas* contain sets of *objects*, such as *tables* and *views*; and *tables* are composed of sets of *columns* and *records*.

Figure 1-1. SQL2003 dataset hierarchy



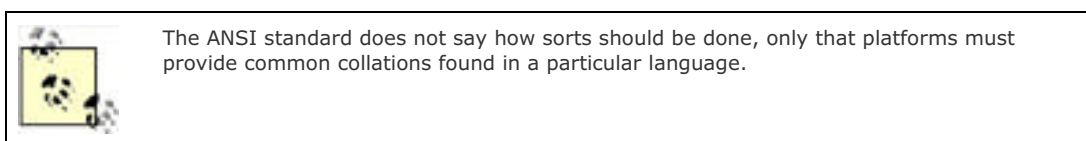
For example, in a **Business_Expense** table, a column called **Expense_Date** might show when an expense was

incurred. Each record in the table describes a specific entity; in this case, everything that makes up a business expense (when it happened, how much it cost, who incurred the expense, what it was for, and so on). Each attribute of an expense, in other words each column, is supposed to be *atomic*; that is, each column is supposed to contain one, and only one, value. If a table is constructed in which the intersection of a row and column can contain more than one distinct value, then one of SQL's primary design guidelines has been violated. (Naturally, some of the database platforms discussed in this book allow you to place more than one value into a column, via *VARRAY* or *TABLE* datatypes.)

There are rules of behavior specified for column values. Foremost is that column values must share a common *domain*, better known as a *datatype*. For example, the value *ELMER* should not be placed into the **Expense_Date** field. The value *ELMER* is a string, not a date, while the **Expense_Date** field can contain only dates. Therefore, this column would be defined as having a *DATE* datatype. In addition, SQL2003 allows further controls of such values through the application of *constraints* and *assertions*. (Constraints are discussed in detail later in [Chapter 2](#).) A SQL constraint might limit **Expense_Date** to expenses less than a year old.

Additionally, data access for all individuals and computer processes is controlled at the schema level by an *AuthorizationID* or *user*. Permissions to specific sets of data may be granted or restricted to each user.

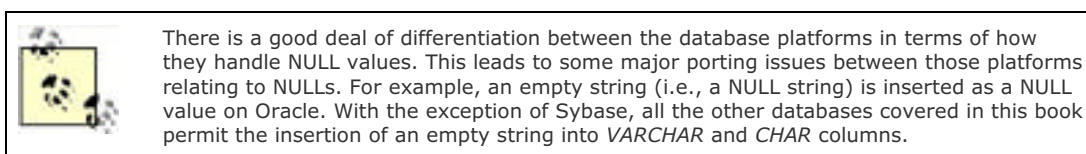
Moreover, SQL databases also employ *character sets* and *collations*. Character sets are the "symbols" or "alphabets" used by the "language" of the data. For example, the American English character set does not contain the special character for ñ in the Spanish character set. Collations are sets of sorting rules that operate on a character set. A collation defines how a given data manipulation operation sorts data. For example, an American English character set might be sorted either by *character-order*, *case-insensitive*, or by *character-order*, *case-sensitive*.



It is important to know what collation you are using when writing SQL code against a database platform since it can have a direct impact on how queries behave, particularly on the behavior of the *WHERE* and *ORDER BY* clauses of *SELECT* statements. For example, a query that sorts data using a binary collation will return data in a very different order than one that sorts data using, say, an American English collation.

1.1.1.2 NULLS (rule 3)

Most databases allow any of their supported datatypes to store NULL values. Inexperienced SQL programmers and developers tend to think of NULL as zero or blank. In fact, NULL is neither of these. In SQL2003, NULL literally means that the value is unknown or indeterminate. (This question alone—whether NULL should be considered unknown or indeterminate—is the subject of academic debate.) This differentiation enables a database designer to distinguish between those entries that represent a deliberately placed zero (for example) and those where either the data is not recorded in the system or where a NULL has been explicitly entered. For an example of this semantic difference, consider a system that tracks payments. A product with a NULL price does not mean that the product is free; instead, a NULL price indicates that the amount is not known or perhaps not yet determined.



One side effect of the indeterminate nature of a NULL value is that it cannot be used in a calculation or a comparison. Here are a few brief but very important rules, from the ANSI standard, to remember about the behavior of NULL values when dealing with NULLs in SQL statements:

- A NULL value cannot be inserted into a column defined as NOT NULL.
- NULL values are not equal to each other. It is a frequent mistake to compare two columns that contain NULL and expect the NULL values to match. (The proper way to identify a NULL value in a *WHERE* clause or in a Boolean expression is to use phrases such as "value IS NULL" and "value IS NOT NULL".)
- A column containing a NULL value is ignored in the calculation of aggregate values such as *AVG*, *SUM*, or *MAX.COUNT*.
- When columns that contain NULL values are listed in the *GROUP BY* clause of a query, the query output

contains a single row for NULL values. In essence, the ANSI standard considers all NULLs found to be in a single group.

- *DISTINCT* and *ORDER BY* clauses, like *GROUP BY*, also see NULL values as indistinguishable from each other. With the *ORDER BY* clause, the vendor is free to choose whether NULL values sort high (first in the result set) or sort low (last in the result set) by default.

1.1.1.3 Metadata (rules 4 and 10)

Codd's fourth rule for relational databases states that data about the database (metadata) must be stored in standard tables, just as all other data. *Metadata* is data that describes the database itself. For example, every time you create a new table or view in a database, records are created and stored that describe the new table. Additional records are needed to store any columns, keys, or constraints on the table. This technique is implemented in most commercial and open source SQL database products. For example, SQL Server uses what it calls "system tables" to track all the information about the databases, tables, and database objects in any given database. It also has "system databases" that keep track of information about the server on which the database is installed and configured.

1.1.1.4 The language (rules 5 and 11)

Codd's rules do *not* require SQL to be used with a relational database. His rules, particularly rules 5 and 11, only specify how the language should behave when coupled with a relational database. At one time, SQL competed with other languages (like Digital's RDO or Fox/PRO) that might have fit the relational bill, but SQL won out for three reasons. First, SQL is a relatively simple, intuitive, English-like language that handles most aspects of data manipulation. Second, SQL is satisfyingly high-level. A programmer or Database Administrator (DBA) does not have to spend time ensuring that data is stored in the proper memory registers or that data is cached to disk. The Database Management System (DBMS) handles that task automatically. Finally, because SQL is not owned by any single vendor, it was adopted across a number of platforms.

1.1.1.5 Views (rule 6)

A *view* is a virtual table that does not exist as a physical data repository of data, but is instead constructed, on the fly, from a *SELECT* statement whenever the view is queried. Views enable you to construct different representations out of the same source data for a variety of audiences without having to alter the way in which the data is stored.



Some vendors support database objects called *materialized views*. Materialized views are not governed by the same rules as ANSI standard views.

1.1.1.6 Set operations (rules 7 and 12)

Other database manipulation languages, such as the venerable Xbase, perform their data operations quite differently from SQL. These languages require you to tell the program exactly how to treat the data, one record at a time. Since the program cycles down through a list of records, performing its logic on one record after another, this style of programming is frequently called *row processing* or *procedural programming*.

In contrast, SQL programs operate on logical *sets* of data. Set theory is applied in most all SQL statements such as *SELECT*, *INSERT*, *UPDATE*, or *DELETE* statements. In effect, data is selected from a set called a table. Unlike the row processing style, *set processing* allows a programmer to tell the database simply *what* is required, not *how* each individual piece of data should be handled. Sometimes set processing is referred to as *declarative processing*, since a programmer declares only what data is wanted, as in "Give me all employees in the southern region who earn more than \$70,000 per year," rather than describing the exact procedure used to retrieve or manipulate the data.



Set theory was the brainchild of mathematician Georg Cantor, who developed it at the end of the nineteenth century. At the time, set theory (and his theory of the infinite) was quite controversial. Today, set theory is such a common part of life that it is learned in elementary school.

Examples of set theory in conjunction with relational databases are detailed in the following section.

1.1.2 Codd's Rules in Action: Simple SELECT Examples

Up to this point, the chapter has been about the individual aspects of a relational database platform as defined by Codd and implemented under ANSI SQL. This section presents a high-level overview of the most important SQL statement, *SELECT*, and some of its most salient points—namely, the relational operations known as *projections*, *selections*, and *joins*:

Projection

Retrieves specific columns of data.

Selection

Retrieves specific rows of data.

Join

Returns columns and rows from two or more tables in a single result set.

Although at first glance it might appear that the *SELECT* statement deals only with the relational selection operation, in actuality, *SELECT* embodies all three operations. (Refer to [SELECT Statement](#).)

The following statement embodies the projection operation by selecting the first and last names of an author, plus his home state, from the **authors** table:

```
SELECT au_fname, au_lname, state
FROM authors
```

The results from any such *SELECT* statement are presented as another table of data:

au_fname	au_lname	state
Johnson	White	CA
Marjorie	Green	CA
Cheryl	Carson	CA
Michael	O'Leary	CA
Meander	Smith	KS
Morningstar	Greene	TN
Reginald	Blotchet-Halls	OR
Innes	del Castillo	MI

The resulting data is sometimes called a *result set*, *work table*, or a *derived table*, differentiating it from the *base table* in the database that is the target of the *SELECT* statement.

It is important to note that the relational operation of projection, and not selection, is specified using the *SELECT* clause (that is, the keyword *SELECT* followed by a list of expressions to be retrieved) of a *SELECT* statement. Selection, the operation of retrieving specific rows of data, is specified using the *WHERE* clause in a *SELECT* statement. *WHERE* filters out unwanted rows of data and retrieves only the requested rows. Using the previous example, the following selects authors from states other than California:

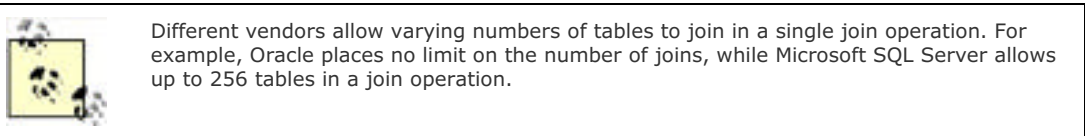
```
SELECT au_fname, au_lname, state
FROM authors
WHERE state <> 'CA'
```

The previous query retrieved all authors; the result of this query is a much smaller subset of records:

au_fname	au_lname	state
Meander	Smith	KS
Morningstar	Greene	TN
Reginald	Blotchet-Halls	OR
Innes	del Castillo	MI

By combining the capabilities of projection and selection together in a single query, you can use SQL to retrieve only the columns and records that you need at any given time.

Joins are the next, and last, relational operation we're going to talk about in this section. A join relates one table to another in order to return a result set consisting of related data from both tables.



The ANSI standard method of performing joins is to use the *JOIN* clause in a *SELECT* statement. An older method, known as a *theta join*, performs the join analysis in the *WHERE* clause. The following example shows both approaches. Each statement retrieves employee information from the **employee** base table as well as job descriptions from the **jobs** base table. The first *SELECT* uses the newer, ANSI *JOIN* clause, while the second *SELECT* uses a theta join:

-- ANSI style

```
SELECT a.au_fname, a.au_lname, t.title_id
FROM authors AS a
JOIN titleauthor AS t ON a.au_id = t.au_id
WHERE a.state <> 'CA'
```

-- Theta style

```
SELECT a.au_fname, a.au_lname, t.title_id
FROM authors AS a,
      titleauthor AS t
WHERE a.au_id = t.au_id
      AND a.state <> 'CA'
```

For more information about joins, refer to [JOIN Subclause](#).



1.2 History of the SQL Standard

Because SQL dialects began to proliferate, the ANSI published its first SQL standard in 1986 to bring greater conformity among vendors, and a second, widely-adopted standard in 1989. The International Standards Organization (ISO) also approves the SQL standard. ANSI released updates in 1992, known as SQL92 and SQL2, and again in 1999, termed both SQL99 and SQL3. Now that the ANSI standard has advanced, we generally use the industry standard term SQL2003. Each time it revises the SQL standard, ANSI adds new features and incorporates new commands and capabilities into the language. For example, the SQL99 standard added a group of capabilities that handled object-oriented datatype extensions.

1.2.1 What's New in SQL2003

SQL99 had two main parts, Foundation:1999 and Bindings:1999. The SQL2003 Foundation section includes all of the Foundation and Bindings standards from SQL99, but a new section called Schemata was created.

The Core requirements of SQL2003 did not change from Core SQL99. So the database platforms that conformed to Core SQL99 automatically conform to SQL2003. Although the Core of SQL2003 has no additions (except for a few new reserved words), a number of individual statements and behaviors have been updated or modified. Because these updates are reflected in the individual syntax descriptions of each statement in [Chapter 3](#), we won't spend time on them here.

A few elements of the Core in SQL99 have been deleted from SQL2003, including:

- The *BIT* and *BIT VARYING* datatypes
- The *UNION JOIN* clause
- The *UPDATE...SET ROW* statement

In addition, a number of features, most of which were rather obscure, have been added, deleted, or renamed. At this point in time, many of the new features of SQL2003 standard are interesting mostly from an academic standpoint because none of the database platforms support the feature yet. However, there are a few new features that hold more than passing interest:

Elementary OLAP functions

Adds an Online Analytical Processing (OLAP) amendment, including a number of windowing functions to support widely used calculations such as moving averages and cumulative sums. Windowing functions are aggregates computed over a window of data: *ROW_NUMBER*, *RANK*, *DENSE_RANK*, *PERCENT_RANK*, and *CUME_DIST*. OLAP functions are fully described in T6.11 of the standard. Some database platforms are starting to support the OLAP functions. Refer to [Chapter 4](#) for details.

Sampling

Adds the *TABLESAMPLE* clause to the *FROM* clause. This is useful for statistical queries on large databases, such as a data warehouse.

Enhanced numeric functions

Adds a large number of numeric functions. In this case, the standard was mostly catching up with the trend in the industry, since the new functions are already supported by one or more of the database platforms. Refer to [Chapter 4](#) for details.

Another important change from SQL92 is that SQL99 is built upon SQL92's *levels of conformance*.

1.2.2 Levels of Conformance

SQL92 first introduced levels of conformance by defining three categories: *Entry*, *Intermediate*, and *Full*. Vendors had to achieve at least Entry-level conformance to claim ANSI SQL compliance. The U.S. National Institute of Standards and Technology (NIST) later added the *Transitional* level between the Entry and Intermediate levels. So NIST's levels of conformance were Entry, Transitional, Intermediate, and Full, while ANSI's were only Entry, Intermediate, and Full. Each higher level of the standard was a superset of the subordinate level, meaning that each higher level of the standard included all the features of the lower level of conformance.

Later, SQL99 altered the base levels of conformance. Gone were the Entry, Intermediate, and Full levels of conformance. With SQL99, vendors must implement all the features of the lowest level of conformance, Core SQL99, in order to claim (and publish) that they are SQL99 ready. Core SQL99 includes the old Entry SQL92 feature set, features from other SQL92 levels, and some brand new features. A vendor may also choose to implement additional feature packages described in the SQL99 standard.

1.2.3 Supplemental Features Packages in the SQL2003 Standard

The SQL2003 standard represents the ideal, but very few vendors currently meet or exceed the Core SQL2003 requirements. The Core standard is like the interstate speed limit: some drivers go above, others go below, but few go exactly the speed limit. Similarly, vendor implementations can vary greatly.

The nine supplemental features packages, representing different subsets of commands, are platform-optional. Some features might show up in multiple packages, while others do not appear in any of the packages. These packages and their features are described in the list that follows.

Two committees—one within ANSI, the other within ISO, and both composed of representatives from virtually every RDBMS vendor—drafted the definitions shown in the list. In this collaborative and somewhat political environment, vendors compromised on exactly which proposed features and implementations were incorporated into a new standard.

Many times, a new feature in the ANSI standard is derived from an existing product or is the outgrowth of new research and development in the academic community. Consequently, vendor adoption of specific ANSI standards can be spotty at times. A relatively new addition to the SQL2003 standard is SQL/XML. The other parts of the SQL99 standard are persisted in SQL2003, though their names may have changed or they may have been slightly rearranged.

Part 1 - SQL/Framework

Includes common definitions and concepts used throughout the standard. Defines the way the standard is structured and how the various parts relate to one another. Describes the conformance requirements set out by the standards committee.

Part 2 - SQL/Foundation

Includes the Core, an augmentation of the SQL99 Core, and is the largest and most important part of the standard.

Part 3 - SQL/CLI (Call-Level Interface)

Defines the call-level interface for dynamically invoking SQL statements from external application programs. SQL/CLI also includes over 60 routine specifications to facilitate the development of truly portable shrink-wrapped software.

Part 4 - SQL/PSM (Persistent Stored Modules)

Standardizes procedural language constructs similar to those found in database platform-specific SQL dialects like PL/SQL and Transact-SQL.

Part 9 - SQL/MED (Management of External Data)

Defines the management of data located outside of the database platform using datalinks and a wrapper interface.

Part 10 - SQL/OBJ (Object Language Binding)

Describes how to embed SQL statements in Java programs. It is closely related to JDBC, but offers a few advantages over JDBC. It is also very different from the traditional host language binding possible in early versions of the standard.

Part 11 - SQL/Schemata

Defines over 85 views (three more than in SQL99) used to describe the metadata of each database and stored in a special schema called *INFORMATION_SCHEMA*. A number of views that existed in SQL99 have been updated.

Part 12 - SQL/JRT (Java Routines and Types)

Defines a number of SQL routines and types using the Java programming language. Features of Java, such as Java static methods and Java classes, are now supported.

Part 14 - SQL/XML

Adds a new type, called XML, four new operators (*XMLPARSE*, *XMLSERIALIZE*, *XMLROOT*, and *XMLCONCAT*), several new functions (described in [Chapter 4](#)), and the new *IS DOCUMENT* predicate. It also includes rules for mapping SQL-related elements (like identifiers, schemas, and objects) to XML-related elements.

Note that parts 5, 6, 7, and 8 do not exist by design.

Be aware that an RDBMS platform may claim SQL2003 compliance by meeting Core SQL99 standards, so read the vendor's fine print for a full description of its ANSI conformity features. By understanding what features comprise the nine packages, you gain a clear idea of both the capabilities of a particular RDBMS and how the various features behave when SQL code is transported to other database products.

The ANSI standards—which cover retrieval, manipulation, and management of data in commands, such as *SELECT*, *JOIN*, *ALTER TABLE*, and *DROP*—formalize many SQL behaviors and syntax structures across a variety of platforms. These standards become even more important as open source database products, such as MySQL and PostgreSQL, grow in popularity and are developed by virtual teams rather than large corporations.

SQL in a Nutshell, Second Edition, explains the SQL implementation of five popular RDBMSs. These vendors do not meet all the SQL2003 standards; in fact, all RDBMS platforms play a constant game of tag with the standards bodies. Many times, as soon as vendors close in on the standard, the standards bodies update, refine, or otherwise change the benchmark. Conversely, the vendors often implement new features that are not yet a part of the standard.

1.2.4 SQL2003 Statement Classes

Comparing statement classes further delineates SQL2003 from SQL92. However, you will still hear these terms bantered about. So you need to know them. In SQL92, SQL statements are grouped into three broad categories:

Data Manipulation Language (DML)

Provides specific data-manipulation commands such as *SELECT*, *INSERT*, *UPDATE*, and *DELETE*.

Data Definition Language (DDL)

Contains commands that handle the accessibility and manipulation of database objects, including *CREATE* and *DROP*.

Data Control Language (DCL)

Contains the permission-related commands *GRANT* and *REVOKE*.

In contrast, SQL2003 supplies seven core categories, now called *classes*, that provide a general framework for the types of commands available in SQL. These statement "classes" are slightly different than the SQL92 statement categories, since they attempt to identify the statements within each class more accurately and logically. Furthermore, because SQL is constantly under development, new features and commands enter the standard and may necessitate new statement classes. So, SQL2003 has a new sets of statement classes, originally added in the SQL99 standard, making them somewhat more comprehensible and logical. Additionally, the new statement classes now allow some "orphaned" statements—which did not fit well into any of the old categories—to be properly classified.

[Table 1-1](#) identifies the SQL2003 statement classes and lists some commands in each class, each of which is fully discussed later. At this point, the key is to remember the statement class title.

Table 1-1. SQL2003 statement classes

Class	Description	Example commands
SQL connection statements	Start and end a client connection	<i>CONNECT</i> , <i>DISCONNECT</i>
SQL control statements	Control the execution of a set of SQL statements	<i>CALL</i> , <i>RETURN</i>
SQL data statements	May have a persistent and enduring effect upon data	<i>SELECT</i> , <i>INSERT</i> , <i>UPDATE</i> , <i>DELETE</i>

SQL diagnostic statements	Provide diagnostic information and raise exceptions and errors	<i>GET DIAGNOSTICS</i>
SQL schema statements	May have a persistent and enduring effect on a database schema and objects within that schema	<i>ALTER, CREATE, DROP</i>
SQL session statements	Control default behavior and other parameters for a session	<i>SET statements like SET CONSTRAINT</i>
SQL transaction statements	Set the starting and ending point of a transaction	<i>COMMIT, ROLLBACK</i>

Those who work with SQL regularly should become familiar with both the old (SQL92) and the new (SQL2003) statement classes, since both nomenclatures are still used to refer to SQL features and statements.

1.3 SQL Dialects

The constantly evolving nature of the SQL standard has given rise to a number of SQL *dialects* among the various vendors and platforms. These dialects most commonly evolved because the user community of a given database vendor required capabilities in the database before the ANSI committee created a standard. Occasionally, though, a new feature is introduced by the academic or research communities due to competitive pressures from competing technologies. For example, many database vendors are augmenting their current programmatic offerings with Java (as is the case with DB2, Oracle, and Sybase) or VBScript (as Microsoft is doing). In the future, programmers and developers will use these programming languages in concert with SQL to build SQL programs.

Many of these dialects include conditional processing capabilities (such as that controlled through *IF . . . THEN* statements), control-of-flow functions (such as *WHILE* loops), variables, and error handling. Because ANSI had not yet developed a standard for these important features at the time users began to demand them, RDBMS developers and vendors were free to create their own commands and syntax. In fact, some of the earliest vendors from the 1980s have variances in the most elementary commands, such as *SELECT*, because their implementations predate the standards. ANSI is now refining standards that address these inconsistencies.

Some of these dialects have introduced procedural commands to support the functionality of a much more complete programming language. For example, these procedural implementations contain error-handling commands, control-of-flow language, conditional commands, variable handling, arrays, and many other extensions. Although these are technically divergent procedural implementations, they are called dialects here. The SQL/PSM (Persistent Stored Module) package provides many features associated with programming stored procedures and incorporates many of the extensions offered by these dialects.

Some popular dialects of SQL include:

PL/SQL

Found in Oracle. PL/SQL stands for Procedural Language/SQL and contains many similarities to the language Ada.

Transact-SQL

Used by both Microsoft SQL Server and Sybase Adaptive Server. As Microsoft and Sybase have moved away from the common platform they shared early in the 1990s, their implementations of Transact-SQL have also diverged.

PL/pgSQL

The name of the SQL dialect and extensions implemented in PostgreSQL. The acronym stands for Procedural Language/postgreSQL.

SQLPL

The newest dialect is DB2's SQLPL (SQLProcedural Language), which is based on the standard SQL control statements. Most of the other dialects predate the standard, meaning you'll find a lot of variations from the SQL standard. But since SQLPL came after the standard, it is in greater compliance.

If you plan to work extensively with a single database system, you should learn the intricacies of your preferred SQL dialect or platform.

Chapter 1. SQL History and Implementations

In the early 1970s, the seminal work of IBM research fellow Dr. E. F. Codd led to the development of a relational data model product called SEQUEL, or Structured English Query Language. SEQUEL ultimately became SQL, or *Structured Query Language*.

IBM, along with other relational database vendors, wanted a standardized method for accessing and manipulating data in a relational database. Although IBM was first to develop relational database theory, Oracle was first to market with the technology. Over time, SQL proved popular enough in the marketplace to attract the attention of the American National Standards Institute (ANSI), which released standards for SQL in 1986, 1989, 1992, 1999, and 2003. Since 1986, competing languages have allowed programmers and developers to access and manipulate relational data. However, few were as easy to learn or as universally accepted as SQL. Programmers and administrators now have the benefit of learning a single language that, with minor adjustments, is applicable to a wide variety of database platforms, applications, and products.

SQL in a Nutshell, Second Edition, describes five common implementations of SQL2003:

- IBM's DB2 Universal Database Version 8 for Linux, Unix, and Windows
- MySQL Version 4
- Oracle Database 10g
- PostgreSQL Version 7
- Microsoft's SQL Server 2000

We also cover Sybase Adaptive Server Enterprise (ASE), but to a lesser degree, in [Appendix B](#).

2.1 Database Platforms Described in This Book

SQL in a Nutshell, Second Edition, describes the SQL standard and the platform-specific implementations of five leading RDBMSs:

DB2

IBM's popular DBMS runs on hardware platforms ranging from PCs all the way up to huge mainframes. It runs on a multitude of operating systems, including the burgeoning Linux OS. DB2 is very popular in corporate environments, especially those with a healthy investment in IBM software, equipment, or services. This book covers DB2 Universal Database Version 8.1 for Linux, Unix, and Windows.

MySQL

MySQL is a popular open source DBMS that is known for its blistering performance. It runs on numerous operating systems, including most Linux variants. To improve performance, it has a slimmer feature set than do many other DBMSs. This book covers MySQL 4.0.

Oracle

Oracle is a leading RDBMS in the commercial sector. It runs on a multitude of operating systems and hardware platforms. Its scalable, reliable architecture has made it the platform of choice for many users. *SQL in a Nutshell*, Second Edition, covers Oracle Database 10g.

PostgreSQL

PostgreSQL is the most feature-rich open source database platform available. Where MySQL is best known for its high-speed performance, PostgreSQL is best known for its excellent support for ANSI standards and robust transaction processing capabilities, as well as rich datatype and database object support. In addition to its full set of features, PostgreSQL runs on a wide variety of operating systems and hardware platforms. This book covers PostgreSQL 7.2.

SQL Server

Microsoft SQL Server is a popular RDBMS that runs only on the Windows platform. Its features include ease of use, an all-inclusive feature set, low cost, and high performance. This book covers Microsoft SQL Server 2000.

In addition to the five databases just mentioned, Sybase Adaptive Server Enterprise (ASE) Version 12.5 is covered separately in [Appendix A](#). Because Sybase and SQL Server have common roots, the appendix discusses the differences between the generally usable SQL Server syntax and those specifics and extensions offered by Sybase ASE.

2.2 Categories of Syntax

To begin to use SQL, you must understand how statements are written. SQL syntax falls into four main categories. Each category is introduced in the following list and then explained in further detail in the sections that follow:

Identifiers

Describe a user- or system-supplied name for database objects, such as a database, a table, a constraint on a table, the columns in a table, a view, etc.

Literals

Describe a user- or system-supplied string or value that is not otherwise an identifier or a keyword. Literals may be strings like "hello", numbers like 1234, dates like "Jan 01, 2002", or Boolean values like TRUE.

Operators

Are symbols specifying an action performed on one or more expressions, most often in *DELETE*, *INSERT*, *SELECT*, or *UPDATE* statements. Operators are also used frequently in the creation of database objects.

Reserved words and keywords

Have special meaning to the database SQL parser, such as *SELECT*, *GRANT*, *DELETE*, or *CREATE*. *Reserved words* are words that cannot be used as identifiers within the database platform, usually commands or SQL statements. *Keywords* are words that may become reserved some time in the future. You can circumvent the restriction on using reserved words and keywords as identifiers through *quoted identifiers*, described in a moment. This is not recommended! Elsewhere in the book (besides here), we use the term *keyword* to describe both concepts.

2.2.1 Identifiers

Keep in mind that RDBMSs are built upon set theory. In ANSI terms, clusters contain sets of catalogs, catalogs contain sets of schemas, schemas contain sets of objects, and so on. Most database platforms use corollary terms: *instances* contain one or more databases; *databases* contain one or more schemas; *schemas* contain one or more tables, views, or stored procedures, and the privileges associated with each object. At each level of this structure, items require a unique name (that is, an identifier) so that they can be referenced by programs and system processes. This means that each *object* (whether a database, table, view, column, index, key, trigger, stored procedure, or constraint) in a RDBMS must be identified. When issuing the command that creates a database object, you must specify an identifier (i.e., a name) for that new object.

There are two important categories of rules that experienced programmers keep in mind when choosing an identifier for a given item:

Naming conventions

Include logical rules of thumb or naming conventions that ultimately create better database structures and data tracking. These are not so much required by SQL as they are the distilled experience of practiced programmers.

Identifier rules

Are those set by the SQL standard and implemented by the platforms. Such rules include characteristics like how long a name may be. These identifier conventions are covered for each vendor later in this chapter.

2.2.1.1 Naming conventions

Naming conventions establish a standard baseline for choosing object identifiers. In this section, we show a list of naming conventions (rules for picking your identifiers) that are based on long years of experience. The SQL standard has no comment on naming conventions outside of the uniqueness of an identifier, its length, and the characters that are valid within the identifier:

Select a name that is meaningful, relevant, and descriptive

Do not name a table **XP03**; instead, name it **Expenses_2005**, showing that it stores expenses for the year 2005. Remember that other people will likely be using the table or database too, perhaps long after you have gone, and the names should make sense at a glance. Each database vendor has limits on object name size, but names generally may be long enough to make sense to anyone reading them.

Choose and apply the same case throughout

Use either all uppercase or all lowercase for all objects throughout the database. Some database servers are case-sensitive, so using mixed-case identifiers might cause problems later.

Use abbreviations consistently

Once an abbreviation has been chosen, it should be used consistently throughout the database. For example, if EMP is used as an abbreviation for EMPLOYEE, then EMP should be used throughout the database. Do not use EMP in some places and EMPLOYEE in others.

Use complete, descriptive, meaningful names with underscores for reading clarity

A column name **UPPERCASEWITHUNDERScores** is not as easy to read as **UPPERCASE_WITH_UNDERScores**.

Do not put company or product names in database object names

Companies get acquired and products change names. These elements are too transitory to be included in database object names.

Do not use overly obvious prefixes or suffixes

For example, don't use "DB_" as a prefix for a database, and don't prefix every view with "V_". Simple queries to the system table of the database can tell the DBA or database programmer what type of object an identifier represents.

Do not fill up all available space for the object name

If the database platform allows a 32-character table name, try to leave at least a few free characters at the end. Some database platforms sometimes append prefixes or suffixes to table names when manipulating temporary copies of the tables.

Do not use quoted identifiers

Quoted identifiers are object names stored within double-quotes. (The ANSI standard calls these *delimited identifiers*.) Encapsulating an identifier within double-quotes allows you to create names that will be difficult to use and that may cause you problems later. Quoted identifiers are also case-sensitive. For example, you could embed spaces, special characters, mixed case characters, or even escape sequences within a quoted identifier, but because some third-party tools (and even vendor-supplied tools) cannot handle special characters in names, you should not make extensive use of quoted identifiers. Some platforms allow other delimiting symbols than double-quotes. For example, SQL Server uses brackets [] to designate quoted identifiers.

There are several benefits to following these naming conventions. First, your SQL code becomes, in a sense, self-documenting because the chosen names are meaningful and understandable to other users. Second, your SQL code and database objects are easier to maintain—especially for other users who come after you—because your objects are consistently named. Finally, maintaining consistency increases database functionality. If the database ever has to be transferred or migrated to another RDMS, consistent and descriptive naming saves both time and energy. Giving a few minutes of thought to naming SQL objects in the beginning can prevent problems later on.

2.2.1.2 Identifier rules

Identifier rules are rules for identifying objects, enforced by the database platforms, within the database. These rules apply to normal identifiers, not quoted identifiers. Rules specified by the SQL2003 standard generally differ somewhat from those of a specific database vendor. [Table 2-1](#) contrasts the SQL2003 rules with those of the five RDMS platforms covered in this book.

Table 2-1. Platform-specific rules for regular object identifiers (excludes quoted identifiers)

Characteristic	Platform	Specification
Identifier size	SQL2003	128 characters
	DB2	128 characters, depending on the object
	MySQL	64 characters
	Oracle	30 bytes (number of characters depends on the character set); database names are limited to 8 bytes
	PostgreSQL	31 characters (NAMEDATALEN property minus 1)
	SQL Server	128 characters (temp tables are limited to 116 characters)
Identifier may contain	SQL2003	Any number, character, or underscore
	DB2	Any number, uppercase character, digit, or the underscore character
	MySQL	Any number, character, or symbol
	Oracle	Any number, character, and the underscore (_), pound (#), and dollar (\$) symbols
	PostgreSQL	Any number, character, or the underscore (_) symbol
	SQL Server	Any number, character, and the underscore (_), at sign (@), pound (#), and dollar (\$) symbols
Identifier must begin with	SQL2003	A letter
	DB2	A letter
	MySQL	A letter or number (but cannot be composed entirely of numbers)
	Oracle	A letter
	PostgreSQL	A letter or underscore (_)
	SQL Server	A letter, underscore (_), at sign (@), or pound (#)
Identifier cannot contain	SQL2003	Spaces or special characters
	DB2	Spaces or special characters
	MySQL	Period (.), slash (/), or ASCII(0) and ASCII(255). Quote (') and double-quote (") are only allowed in quoted identifiers.
	Oracle	Spaces, double-quotes ("), or special characters
	PostgreSQL	Double-quote (")
	SQL Server	Spaces or special characters
Allows quoted identifiers	SQL2003	Yes
	DB2	Yes
	MySQL	Yes
	Oracle	Yes
	PostgreSQL	Yes
	SQL Server	Yes
Quoted identifier symbol	SQL2003	Double-quote (")
	DB2	Double-quote (")
	MySQL	Quote (') or double-quote (") in ANSI compatibility mode

	Oracle	Double-quote (")
	PostgreSQL	Double-quote (")
	SQL Server	Double-quote (") or brackets ([]); brackets are preferred
Identifier may be reserved	SQL2003	No, unless as a quoted identifier
	DB2	Yes
	MySQL	No, unless as a quoted identifier
	Oracle	No, unless as a quoted identifier
	PostgreSQL	No, unless as a quoted identifier
	SQL Server	No, unless as a quoted identifier
Schema addressing	SQL2003	<i>Catalog.schema.object</i>
	DB2	Schema.object
	MySQL	Database.object
	Oracle	Schema.object
	PostgreSQL	Database.schema.object
	SQL Server	Server.database.schema.object
Identifier must be unique	SQL2003	Yes
	DB2	Yes
	MySQL	Yes
	Oracle	Yes
	PostgreSQL	Yes
	SQL Server	Yes
Other rules	SQL2003	None
	DB2	None
	MySQL	May not contain numbers only
	Oracle	Database links are limited to 128 bytes and may not be quoted identifiers
	PostgreSQL	None
	SQL Server	None

Identifiers must be unique within their scope. Thus, in our earlier discussion of the hierarchy of database objects, database names must be unique on a given instance of a database server, while the names of tables, views, functions, triggers, and stored procedures must be unique within a given schema. On the other hand, you *can* have a table and a stored procedure with the same name since they are different object types. The names of columns, keys, and indexes must be unique on a single table or view, and so forth. Check with the database platform documentation for more information — some platforms require unique identifiers where others may not. For example, DB2 requires all index identifiers to be unique throughout the database, while SQL Server requires that the index identifier be unique only for the table it depends on.

Remember, quoted identifiers (object names encapsulated within a special delimiter, usually double quotes) may be used to break some of the identifier rules specified earlier. Specifically, quoted identifiers may be used to bestow a reserved word as a name, or to allow normally unusable characters and symbols within a name. For example, you normally can't use the percent sign (%) in a table name. However, you can, if you must, use the percent sign in a table name so long as you always enclose that table name within double quotes. To name a table **expense%%ratios**, you would specify the name in quotes as "**expense%%ratios**". Again, remember that in SQL2003, such names are sometimes known as delimited identifiers.



Once you have created an object name as a quoted identifier, we recommend that you always reference it using its special delimiter.

2.2.2 Literals

SQL evaluates literal values as any explicit numeric, character string, temporal value (like a date or time), or Boolean value that is not an identifier or a keyword. SQL databases allow a variety of literal values in a SQL program. Literal values are allowed for most of the numeric, character, Boolean, and date datatypes. For example, SQL Server numeric datatypes include (among others) *INTEGER*, *REAL*, and *MONEY*. Thus, *numeric literals* can look like:

30

-117

+883.3338

-6.66

\$70000

2E5

7E-3

As the example illustrates, SQL Server allows signed or unsigned numerals, in scientific or normal notation. And since SQL Server has a money datatype, even a dollar sign can be included. SQL Server does not allow other symbols in numeric literals (besides 0 1 2 3 4 5 6 7 8 9 + - \$. E e), so do not include commas (or periods in Europe). Most databases interpret a comma in a numeric literal as a *list item separator*. Thus, the literal value 3,000 would be interpreted by the database as 3 and, separately, 000.

Boolean, character string, and date literals look like:

TRUE

'Hello world!'

'OCT-28-1966 22:14:30:00'

Character string literals should always be enclosed by single quotation marks (' '), the standard delimiter for all character string literals. Character string literals are not restricted just to the alphabet. In fact, any character in the character set can be represented as a string literal. All of the following are string literals:

'1998'

'70,000 + 14000'

'There once was a man from Nantucket,'

'Oct 28, 1966'

All of these examples are, in fact, compatible with the *CHARACTER* datatype. Remember not to confuse the string literal `1998' with the numeric literal 1998. Once string literals are associated with *CHARACTER* datatypes, it is poor practice to use them in arithmetic operations without explicitly converting them to a numeric datatype. Some database products will perform automatic conversion of string literals containing numbers when compared against any *DATE* or *NUMBER* datatype values.

By doubling the delimiter, you can effectively represent a single quotation mark in a literal string, if necessary. That is, use two quotation marks each time a single quotation mark is part of the value. This example taken from SQL Server illustrates the idea:

SELECT 'So he said "Who"s Le Petomaine?'"

This gives the result:

So he said 'Who's Le Petomaine?'

2.2.3 Operators

An operator is a symbol specifying an action that is performed on one or more expressions. Operators are used most often in *DELETE*, *INSERT*, *SELECT*, or *UPDATE* statements, but also are used frequently in the creation of database objects, such as stored procedures, functions, triggers, and views.

Operators typically fall into these categories:

Arithmetic operators

Supported by all databases

Assignment operators

Supported by all databases

Bitwise operators

Supported by Microsoft SQL Server

Comparison operators

Supported by all databases

Logical operators

Supported by DB2, Oracle, SQL Server, and PostgreSQL

Unary operators

Supported by DB2, Oracle, and SQL Server

2.2.3.1 Arithmetic operators

Arithmetic operators perform mathematical operations on two expressions of any datatypes in the numeric datatype category. See [Table 2-2](#) for a listing of the arithmetic operators.

Table 2-2. Arithmetic operators

Arithmetic operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modula (SQL Server only); returns the remainder of a division operation as an integer value



In DB2, Oracle, and SQL Server, the + and - operators also can be used to perform arithmetic operations on date values.

2.2.3.2 Assignment operators

Except in Oracle, which uses :=, the assignment operator (=) assigns a value to a variable or the alias of a column heading. In SQL Server, the keyword AS may serve as an operator for assigning table- or column-heading aliases.

2.2.3.3 Bitwise operators

Microsoft SQL Server provides bitwise operators as a shortcut to perform bit manipulations between two-integer expressions (see [Table 2-3](#)). Valid datatypes that are accessible to bitwise operators include *binary*, *bit*, *int*, *smallint*, *tinyint*, and *varbinary*.

Table 2-3. Bitwise operators

Bitwise operator	Meaning
&	Bitwise <i>AND</i> (two operands)
	Bitwise <i>OR</i> (two operands)
^	Bitwise exclusive <i>OR</i> (two operands)

2.2.3.4 Comparison operators

Comparison operators test whether two expressions are equal or unequal. The result of a comparison operation is a Boolean value: *TRUE*, *FALSE*, or *UNKNOWN*. Also, note that the ANSI standard behavior for a comparison operation where one or more of the expressions are NULL is NULL. For example, the expression *23 + NULL* returns NULL, as does the expression *Feb 23, 2002 + NULL*. See [Table 2-4](#) for a list of the comparison operators.

Table 2-4. Comparison operators

Comparison operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
!=	Not equal to (not ANSI standard)
!<	Not less than (not ANSI standard)
!>	Not greater than (not ANSI standard)

Boolean comparison operators are used most frequently in a *WHERE* clause to filter the rows that qualify for the search conditions. The following Microsoft SQL Server example uses the greater than or equal to comparison operation:

```
SELECT *  
  
FROM Products  
  
WHERE ProductID >= @MyProduct
```

2.2.3.5 Logical operators

Logical operators are commonly used in a *WHERE* clause to test for the truth of some condition. Logical operators return a Boolean value of either *TRUE* or *FALSE*. Logical operators also are discussed in [SELECT Statement](#). Not all database systems support all operators. See [Table 2-5](#) for a list of logical operators.

Table 2-5. Logical operators

Logical operator	Meaning
ALL	<i>TRUE</i> if all of a set of comparisons are <i>TRUE</i>
AND	<i>TRUE</i> if both Boolean expressions are <i>TRUE</i>
ANY	<i>TRUE</i> if any one of a set of comparisons is <i>TRUE</i>
BETWEEN	<i>TRUE</i> if the operand is within a range
EXISTS	<i>TRUE</i> if a subquery contains any rows
IN	<i>TRUE</i> if the operand is equal to one of a list of expressions or one or more rows returned by a subquery
LIKE	<i>TRUE</i> if the operand matches a pattern
NOT	Reverses the value of any other Boolean operator
OR	<i>TRUE</i> if either Boolean expression is <i>TRUE</i>
SOME	<i>TRUE</i> if some of a set of comparisons is <i>TRUE</i>

2.2.3.6 Unary operators

Unary operators perform an operation on only one expression of any of the datatypes of the numeric datatype category. Unary operators may be used on integer datatypes, though positive and negative may be used on any numeric datatype (see [Table 2-6](#)).

Table 2-6. Unary operators

Unary operator	Meaning
+	Numeric value is positive
-	Numeric value is negative
~	A bitwise NOT, returns the complement of the number (not in Oracle or DB2)

2.2.3.7 Operator precedence

Sometimes operator expressions become rather complex. When an expression has multiple operators, *operator precedence* determines the sequence in which the operations are performed. The order of execution can significantly affect the resulting value.

Operators have the precedence levels listed below. An operator on a higher level is evaluated before an operator on a lower level. The following listing denotes operators from highest to lowest precedence:

- () (parenthetical expressions)
- +, -, ~ (unary operators)
- *, /, % (mathematical operators)
- +, - (arithmetic operators)

- =, >, <, >=, <=, <>, !=, !>, !< (comparison operators)
- ^ (Bitwise Exclusive OR), & (Bitwise AND), | (Bitwise OR)
- NOT
- AND
- ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
- = (variable assignment)

Operators are evaluated from left to right when they are of equal precedence. However, parentheses are used to override the default precedence of the operators in an expression. Expressions within a parentheses are evaluated first, while operations outside the parentheses are evaluated next.

For example, the following expressions in an Oracle query return very different results:

```
SELECT 2 * 4 + 5 FROM dual
```

```
-- Evaluates to 8 + 5 which yields an expression result of 13.
```

```
SELECT 2 * (4 + 5) FROM dual
```

```
-- Evaluates to 2 * 9 which yields an expression result of 18.
```

In expressions with nested parentheses, the most deeply nested expression is evaluated first.

This next example contains nested parentheses, with the expression 5 - 3 in the most deeply nested set of parentheses. This expression yields a value of 2. Then the addition operator (+) adds this result to 4, which yields a value of 6. Finally, the 6 is multiplied by 2 to yield an expression result of 12:

```
SELECT 2 * (4 + (5 - 3) ) FROM dual
```

```
-- Evaluates to 2 * (4 + 2) which further evaluates to 2 * 6,
```

```
--and yields an expression result of 12.
```

```
RETURN
```

2.2.3.8 System delimiters and operators

String delimiters mark the boundaries of a string of alphanumeric characters. *System delimiters* are those symbols within the character set that have special significance to your database server. *Delimiters* are symbols that are used to judge the order or hierarchy of processes and list items. *Operators* are those delimiters used to judge values in comparison operations, including symbols commonly used for arithmetic or mathematical operations. [Table 2-7](#) lists the system delimiters and operators allowed by SQL.

Table 2-7. SQL delimiters and operators

Symbol	Usage	Example
+	Addition operator; in SQL Server, also serves as a concatenation operator	On all database platforms: <pre>SELECT MAX(emp_id) + 1 FROM employee</pre>
		As a subtraction operator: <pre>SELECT MIN(emp_id) - 1 FROM employee</pre>


-	Subtraction operator; also a range indicator in <i>CHECK</i> constraints	As a range operator, in a <i>CHECK</i> constraint: ALTER TABLE authors ADD CONSTRAINT authors_zip_num CHECK (zip LIKE '%[0-9]%)'
*	Multiplication operator	SELECT salary * 0.05 AS 'bonus' FROM employee;
/	Division operator	SELECT salary / 12 AS 'monthly' FROM employee;
=	Equality operator	SELECT * FROM employee WHERE lname = 'Fudd'
<>	Inequality operators (!= is a nonstandard equivalent on several platforms)	On all platforms: SELECT * FROM employee WHERE lname <> 'Fudd'
< <=	Less-than operator Less-than or equal-to operator	SELECT lname, emp_id, (salary * 0.05) AS bonus FROM employee WHERE (salary * 0.05) <= 10000 AND exempt_status < 3
> >=	Greater-than operator Greater-than or equal-to operator	SELECT lname, emp_id, (salary * 0.025) AS bonus FROM employee WHERE (salary * 0.025) > 10000 AND exempt_status >= 4
()	Expression, function calls, order of operations, and subquery delimiter	Expression: SELECT (salary / 12) AS monthly FROM employee WHERE exempt_status >= 4 Function call:

		<pre>SELECT SUM(travel_expenses) FROM "expense%%ratios"</pre>
		<pre>Order of operations: SELECT (salary / 12) AS monthly, ((salary / 12) / 2) AS biweekly FROM employee WHERE exempt_status >= 4</pre>
		<pre>Subquery: SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')</pre>
%	Wildcard attribute indicator (refer to LIKE Operator .)	<pre>SELECT * FROM employee WHERE lname LIKE 'Fud%'</pre>
,	List item separator	<pre>SELECT lname, fname, ssn, hire_date FROM employee WHERE lname = 'Fudd'</pre>
.	Identifier qualifier separator	<pre>SELECT * FROM scott.employee WHERE lname LIKE 'Fud%'</pre>
'	Character string indicators	<pre>SELECT * FROM employee WHERE lname LIKE 'FUD%' OR fname = 'ELMER'</pre>

"	Quoted identifier indicators	<pre>SELECT expense_date, SUM(travel_expense) FROM "expense%%ratios" WHERE expense_date BETWEEN '01-JAN-2004' AND '01-APR-2004'</pre>
—	Single-line comment delimiter (two dashes followed by a space)	<pre>-- Finds all employees like Fudd, -- Fudge, and Fudston SELECT * FROM employee WHERE Iname LIKE 'Fud%'</pre>
/* */	Beginning multiline comment delimiter Ending multiline comment indicator	<pre>/* Finds all employees like Fudd, Fudge, and Fudston */ SELECT * FROM employee WHERE Iname LIKE 'Fud%'</pre>

2.2.4 Keywords and Reserved Words

Just as certain symbols have special meaning and functionality within SQL, certain words and phrases have special significance. SQL *keywords* are words whose meanings are so closely tied to the operation of the RDBMS that they should not be used for any other purpose; generally, they are words used in a SQL statement. (Note that they *can* be used as an identifier on most platforms, but they shouldn't be.) For example, the word "SELECT" is a reserved word and should not be used as a table name.



It is generally a good idea to avoid naming columns or tables after a keyword that occurs in *any* major platform, because database applications are frequently converted from one platform to another.

Reserved words, on the other hand, do not have special significance now, but they probably will in a future release. To emphasize the fact that keywords should not be used as an identifier, but nevertheless could be, the SQL standard calls them "nonreserved keywords." Reserved words and keywords are not always words used in SQL statements, but may be words commonly associated with database technology. For example, *CASCADE* is used to describe data manipulations that allow their action, such as a delete or update, to "flow down," or cascade, to any subordinate tables. Reserved words and keywords are widely published so that programmers will not use them as identifiers that will, at some later revision, cause a problem.

SQL2003 specifies its own list of reserved words and keywords. In addition, the database platform specify their own list of reserved words and keywords because they each have their own extensions to the SQL command set. SQL standard keywords, as well as the keywords in the different vendor implementations, are shown in [Appendix B](#).

2.3 SQL2003 and Platform-Specific Datatypes

A table can contain one or many columns. Each column must be defined with a datatype that provides a general classification of the data that the column will store. In real-world applications, datatypes provide some control and efficiency as to how tables are defined and how the data is stored within the table. Using specific datatypes enables better, more understandable queries and helps control the integrity of data.

The tricky thing about SQL2003 datatypes is that they do not always map directly to an identical implementation in a given platform. Although the platforms specify "datatypes" that correspond to the SQL2003 datatypes, these are not always true SQL2003 datatypes. For example, MySQL's implementation of a *BIT* datatype is actually identical to a *CHAR(1)* datatype value. Nonetheless, each of the platform datatypes is close enough to the standard to be both easily understandable and job-ready.

The official SQL2003 datatypes (as opposed to platform-specific datatypes) fall into the general categories described in [Table 2-8](#). (Note that the SQL2003 standard contains a few rarely used datatypes (*ARRAY*, *MULTISET*, *REF*, and *ROW*) that are shown only in [Table 2-8](#) and not elsewhere in the book.)

Table 2-8. SQL2003 categories and datatypes

Category	Example datatypes and abbreviations	Description
BINARY	<i>BINARY LARGE OBJECT (BLOB)</i>	This datatype stores binary string values in hexadecimal format. Binary string values are stored without reference to any character set and without any length limit.
BOOLEAN	<i>BOOLEAN</i>	This datatype stores truth values—either <i>TRUE</i> or <i>FALSE</i> .
CHARACTER string types	<i>CHAR</i> <i>CHARACTER VARYING (VARCHAR)</i>	These datatypes can store any combination of characters from the applicable character set. The varying datatypes allow variable lengths, while the other datatypes allow only fixed lengths. Also, the variable-length datatypes automatically trim trailing spaces, while the other datatypes pad all open space.
	<i>NATIONAL CHARACTER (NCHAR)</i> <i>NATIONAL CHARACTER VARYING (NCHAR VARYING)</i>	The national character datatypes are designed to support a particular implementation-defined character set.
	<i>CHARACTER LARGE OBJECT (CLOB)</i>	<i>CHARACTER LARGE OBJECT</i> and <i>BINARY LARGE OBJECT</i> are collectively referred to as <i>large object string types</i> .
	<i>NATIONAL CHARACTER LARGE OBJECT (NCLOB)</i>	Same as <i>CHARACTER LARGE OBJECT</i> , but supports a particular implementation-defined character set.
DATALINK	<i>DATALINK</i>	Defines a reference to a file or other external data source that is not part of the SQL environment.
INTERVAL	<i>INTERVAL</i>	Specifies a set of time values or span of time.
COLLECTION	<i>ARRAY</i> <i>MULTISET</i>	<i>ARRAY</i> was offered in SQL99. <i>MULTISET</i> was added in SQL2003. Whereas an <i>ARRAY</i> is a set-length, ordered collection of elements, <i>MULTISET</i> is a variable-length, unordered collection of elements. The elements <i>ARRAY</i> and <i>MULTISET</i> must be of a predefined datatype.
NUMERIC	<i>INTEGER (INT)</i> <i>SMALLINT</i> <i>BIGINT</i> <i>NUMERIC(p,s)</i> <i>DEC[IMAL] (p,s)</i>	These datatypes store exact numeric values (integers or decimals) or approximate (floating point) values. <i>INT</i> , <i>BIGINT</i> , and <i>SMALLINT</i> store exact numeric values with a predefined precision and a scale of zero. <i>NUMERIC</i> and <i>DEC</i> store exact numeric values with a definable precision and a definable scale. <i>FLOAT</i> stores approximate numeric values with a definable precision, while <i>REAL</i> and <i>DOUBLE PRECISION</i> have predefined precisions. You may define a precision (p) and scale (s) for a <i>DECIMAL</i> , <i>FLOAT</i> , or <i>NUMERIC</i> datatype to indicate the total number of allowed digits in the number and the number of decimal places, respectively.

	<p><i>FLOAT(p,s)</i></p> <p><i>REAL</i></p> <p><i>DOUBLE PRECISION</i></p>	<p><i>INTEGER (INT)</i>, <i>SMALLINT</i>, and <i>DEC[IMAL]</i> (p,s) are sometimes referred to as <i>exact numeric types</i>, while <i>FLOAT (p,s)</i>, <i>REAL</i>, and <i>DOUBLE PRECISION</i> are sometimes called <i>approximate numeric types</i>.</p>
TEMPORAL	<p><i>DATE</i></p> <p><i>TIME</i></p> <p><i>TIME WITH TIME ZONE</i></p> <p><i>TIMESTAMP</i></p> <p><i>TIMESTAMP WITH TIME ZONE</i></p>	<p>These datatypes handle values related to time. <i>DATE</i> and <i>TIME</i> are self-explanatory. Datatypes with the <i>WITH TIME ZONE</i> suffix also include a time zone offset. The <i>TIMESTAMP</i> datatypes store a value that represents the a precise moment in time. Temporal types are also known as <i>datetime types</i>.</p>
XML	<i>XML</i>	<p>Stores XML data and can be used wherever a SQL datatype is allowed, such as a column of a table, field in a row, etc. Operations on the values of an XML type assume a tree-based internal data structure. The internal data structure is based on the XML Information Set Recommendation (Infoset) using a new document information item called the XML root information item.</p>

Not every database platform supports each and every ANSI SQL datatype. [Table 2-9](#) compares datatypes across the five platforms. The table is organized by datatype name. Be careful to look for footnotes when reading this table because some platforms support a datatype of a given name, but implement it in a different or contrary way than the ANSI standard and/or other vendors.


	<p>While platforms may support similarly named datatypes, their implementations can vary. Please consult the detailed information in this chapter for specific requirements of each platform's datatypes.</p>
--	---

Table 2-9. Comparison of platform-specific datatypes

Vendor datatype	DB2	MySQL	Oracle	PostgreSQL	SQL Server	SQL2003 datatype
<i>BFILE</i>			Y			none
<i>BIGINT</i>	Y	Y			Y	<i>BIGINT</i>
<i>BINARY</i>					Y	<i>BLOB</i>
<i>BINARY_FLOAT</i>			Y			<i>FLOAT</i>
<i>BINARY_DOUBLE</i>			Y			<i>DOUBLE PRECISION</i>
<i>BIT</i>		Y		Y	Y	none
<i>BIT VARYING, VARBIT</i>				Y		none
<i>BLOB</i>	Y	Y	Y			<i>BLOB</i>
<i>BOOL, BOOLEAN</i>		Y		Y		<i>BOOLEAN</i>
<i>BOX</i>				Y		none
<i>BYTEA</i>				Y		<i>BLOB</i>
<i>CHAR, CHARACTER</i>	Y	Y	Y	Y	Y	<i>CHARACTER</i>
<i>CHAR FOR BIT DATA</i>	Y					none
<i>CIDR</i>				Y		none
<i>CIRCLE</i>				Y		none

<i>CLOB</i>	Y		Y			<i>CLOB</i>
<i>CURSOR</i>					Y	none
<i>DATALINK</i>	Y					<i>DATALINK</i>
<i>DATE</i>	Y	Y	Y	Y		<i>DATE</i>
<i>DATETIME</i>		Y			Y	<i>TIMESTAMP</i>
<i>DBCLOB</i>	Y					<i>NCLOB</i>
<i>DEC, DECIMAL</i>	Y	Y	Y	Y	Y	<i>DECIMAL</i>
<i>DOUBLE, DOUBLE PRECISION</i>	Y	Y	Y	Y ^[1]	Y	<i>FLOAT</i>
<i>ENUM</i>		Y				none
<i>FLOAT</i>	Y	Y	Y		Y	<i>DOUBLE PRECISION</i>
<i>FLOAT4</i>				Y ^[2]		<i>FLOAT(P)</i>
<i>FLOAT8</i>				Y ^[3]		<i>FLOAT(P)</i>
<i>GRAPHIC</i>	Y					<i>BLOB</i>
<i>IMAGE</i>					Y	none
<i>INET</i>				Y		none
<i>INT, INTEGER</i>	Y	Y	Y	Y	Y	<i>INTEGER</i>
<i>INT2</i>				Y		<i>SMALLINT</i>
<i>INT4</i>				Y		<i>INT, INTEGER</i>
<i>INTERVAL</i>				Y		<i>INTERVAL</i>
<i>INTERVAL DAY TO SECOND</i>			Y			<i>INTERVAL DAY TO SECOND</i>
<i>INTERVAL YEAR TO MONTH</i>			Y			<i>INTERVAL YEAR TO MONTH</i>
<i>LINE</i>				Y		none
<i>LONG</i>			Y			none
<i>LONG VARCHAR</i>	Y					none
<i>LONGBLOB</i>		Y				<i>BINARY LARGE OBJECT</i>
<i>LONG RAW</i>			Y			<i>BLOB</i>
<i>LONG VARGRAPHIC</i>	Y					none
<i>LONGTEXT</i>		Y				none
<i>LSEG</i>				Y		none
<i>MACADDR</i>				Y		none
<i>MEDIUMTEXT</i>		Y				none
<i>MEDIUMBLOB</i>		Y				none
<i>MONEY</i>				Y ^[4]	Y	none
<i>NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, NCHAR VARYING, NVARCHAR</i>		Y	Y		Y	<i>NATIONAL CHARACTER VARYING</i>

<i>NCHAR, NATIONAL CHAR, NATIONAL CHARACTER</i>			Y		Y	<i>NATIONAL CHARACTER</i>
<i>NCLOB</i>			Y			<i>NCLOB</i>
<i>NTEXT, NATIONAL TEXT</i>					Y	<i>NCLOB</i>
<i>NVARCHAR2(N)</i>			Y			none
<i>NUMBER</i>	Y	Y	Y	Y ^[5]	Y	none
<i>NUMERIC</i>						<i>NUMERIC</i>
<i>OID</i>				Y		none
<i>PATH</i>				Y		none
<i>POINT</i>				Y		none
<i>POLYGON</i>				Y		none
<i>RAW</i>			Y			none
<i>REAL</i>	Y	Y	Y		Y	<i>REAL</i>
<i>ROWID</i>			Y			none
<i>ROWVERSION</i>					Y	none
<i>SERIAL, SERIAL4</i>				Y		none
<i>SERIAL8, BIGSERIAL</i>				Y		none
<i>SET</i>		Y				none
<i>SMALLDATETIME</i>					Y	none
<i>SMALLINT</i>	Y	Y	Y		Y	<i>SMALLINT</i>
<i>SMALLMONEY</i>					Y	none
<i>SQL_VARIANT</i>					Y	none
<i>TABLE</i>					Y	none
<i>TEXT</i>		Y		Y	Y	none
<i>TIME</i>	Y			Y		<i>TIME</i>
<i>TIMESPAN</i>				Y		<i>INTERVAL</i>
<i>TIMESTAMP</i>	Y		Y	Y	Y	<i>TIMESTAMP</i>
<i>TIMETZ</i>				Y		<i>TIME WITH TIME ZONE</i>
<i>TINYINT</i>					Y	none
<i>UNIQUEIDENTIFIER</i>					Y	none
<i>UROWID</i>			Y			none
<i>VARBINARY</i>					Y	<i>BLOB</i>
<i>VARCHAR, CHAR VARYING, CHARACTER VARYING</i>	Y	Y	Y ^[6]	Y	Y	<i>CHARACTER VARYING(N)</i>
<i>VARCHAR2</i>			Y			<i>CHARACTER VARYING</i>
<i>VARCHAR FOR BIT DATA</i>	Y					<i>BIT VARYING</i>
<i>VARGRAPHIC</i>	Y					<i>NCHAR VARYING</i>
<i>XMLTYPE</i>			Y			<i>XML</i>

[1] Synonym for *FLOAT*

[2] Synonym for *REAL*

[3] Synonym for *DOUBLE PRECISION*

[4] Synonym for *DECIMAL(9,2)*

[5] Synonym for *DECIMAL*

[6] Oracle vastly prefers *VARCHAR2*

Each of the following sections lists platform-specific datatypes, the SQL2003 datatype category (if any), and pertinent details. Descriptions are provided for non-SQL2003 datatypes.

2.3.1 DB2 Datatypes

As shown below, DB2 supports a rich variety of datatypes, including most of the SQL2003 datatypes:

BIGINT (SQL2003 Datatype: *BIGINT*)

Stores signed or unsigned integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.
Uses 8 bytes of storage.

BLOB (SQL2003 Datatype: *BLOB*)

Holds variable-length binary data up to 2,147,483,647 bytes in length.

CHAR(n), *CHARACTER(n)* (SQL2003 Datatype: *CHARACTER(n)*)

Holds fixed-length character data up to 254 bytes in length. Uses (*n*) bytes of storage.

CHAR(n) FOR BIT DATA (SQL2003 Datatype: *none*)

Holds a fixed number of values. Uses (*n*) bytes of storage.

CLOB (SQL2003 Datatype: *CLOB*)

Holds variable-length character data up to 2,147,483,647 bytes in length.

DATALINK(n) (SQL2003 Datatype: *DATALINK*)

Stores a link to a file external to the database. Uses $n+54$ bytes of storage.

DATE (SQL2003 Datatype: *DATE*)

Holds a calendar date without time of day. Uses 4 bytes of storage.

DBCLOB(n) (SQL2003 Datatype: *NCLOB*)

Holds variable-length double-byte character data up to 107,3741,823 characters in length.

DEC(p,s), *DECIMAL(p,s)* (SQL2003 Datatype: *DECIMAL*, *DECIMAL(p,s)*)

May have a precision of 1 to 31 and a scale of 0 to 31. Uses the integral part of $(p/2)+1$, where *p* is the

precision bytes storage.

DOUBLE, DOUBLE PRECISION (SQL2003 Datatype: *DOUBLE PRECISION*)

Holds floating point numbers of -1.79769E+308 through 1.79769E+308. Uses 8 bytes of storage.

FLOAT (SQL2003 Datatype: *FLOAT*)

A synonym of *DOUBLE PRECISION*.

FLOAT(p) (SQL2003 Datatype: *REAL, DOUBLE PRECISION*)

Can range from 1 through 53. If $p \leq 24$, then *FLOAT(p)* is a synonym for *REAL*. If $24 < p \leq 53$ then *FLOAT(p)* is a synonym for *DOUBLE PRECISION*.

GRAPHIC(n) (SQL2003 Datatype: *NATIONAL CHARACTER*)

Is not an image datatype, but rather holds fixed-length character strings (DBCS) up to 127 characters in length. Uses $n*2$ bytes of storage when using double-byte character sets or n bytes of storage when using single-byte character sets.

INT, INTEGER (SQL2003 Datatype: *INTEGER*)

Stores signed or unsigned integers between -2,147,483,648 and 2,147,483,647. Uses 4 bytes of storage.

LONG VARCHAR (SQL2003 Datatype: *VARCHAR*)

Holds variable-length character data up to 32,700 bytes in length. Uses 24 bytes of storage.

LONG VARCHAR FOR BIT DATA (SQL2003 Datatype: *BIT VARYING*)

Holds variable-length character data up to 32,700 bytes in length. Uses 24 bytes of storage.

LONG VARGRAPHIC (SQL2003 Datatype: none)

Holds variable-length double-byte character strings (DBCS) up to 16,350 characters in length. Uses 24 bytes of storage.

NUM(p,s), NUMERIC(p,s) (SQL2003 Datatype: *NUMERIC, NUMERIC(p,s)*)

Is a synonym for *DECIMAL*. Uses the integral part of $(p/2)+1$, where p is the precision bytes storage.

REAL (SQL2003 Datatype: *REAL*)

Holds floating point numbers of -3.402E+38 through 3.402E+38. Uses 4 bytes of storage.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

Stores signed or unsigned integers between -32,768 and 32,767. Uses 2 bytes of storage.

TIME (SQL2003 Datatype: *TIME*)

Holds the time of day. Uses 3 bytes of storage.

TIMESTAMP (SQL2003 Datatype: *TIMESTAMP*)

Stores the date and time. Uses 10 bytes of storage.

VARCHAR(n), *CHAR VARYING(n)*, *CHARACTER VARYING(n)* (SQL2003 Datatype: *CHARACTER VARYING*, *CHARACTER VARYING(n)*)

Holds variable-length character data up to 32,672 bytes in length. Uses $n+4$ bytes of storage.

VARCHAR(n) FOR BIT DATA (SQL2003 Datatype: *none*)

Stores a variable number of values. Uses $n+4$ bytes of storage.

VARGRAPHIC(n) (SQL2003 Datatype: *NCHAR VARYING*)

Holds variable-length double-byte character data up to 16,336 characters in length. Uses $(n*2)+4$ bytes of storage.

2.3.2 MySQL Datatypes

MySQL numeric datatypes support the following optional attributes:

UNSIGNED

The numeric value is assumed to be non-negative (positive or zero). For fixed-point datatypes like *DECIMAL* and *NUMERIC*, the space normally used to show a positive or negative condition of the numeric value can then be used as part of the value, providing a little extra numeric range in the column for *DECIMAL* and *NUMERIC* types. (There is no *SIGNED* optional attribute.)

ZEROFILL

Used for display formatting, this attribute tells MySQL that the numeric value is padded with zeros to its full size rather than spaces. *ZEROFILL* automatically forces the *UNSIGNED* attribute as well.

MySQL also enforces a maximum display size for columns of up to 255 characters. Columns longer than 255 characters are stored properly, but only 255 characters are displayed. Floating-point numeric datatypes may have a maximum of 30 digits after the decimal point.

The list below shows that MySQL supports most of the SQL2003 datatypes, plus several additional datatypes used to contain lists of values, as well as datatypes used for binary large objects (*BLOBs*). Datatypes that extend the ANSI standard include *TEXT*, *ENUM*, *SET*, and *MEDIUMINT*. Special datatype attributes that go beyond the ANSI standard include *AUTO_INCREMENT*, *BINARY*, *NULL*, *UNSIGNED*, and *ZEROFILL*.

BIGINT(n) [*UNSIGNED*] [*ZEROFILL*] (SQL2003 Datatype: *BIGINT*)

Stores signed or unsigned integers. The signed range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The unsigned range is 0 to 18,446,744,073,709,551,615. *BIGINT* can perform imprecise calculations due to rounding issues.

BIT, *BOOL* (SQL2003 Datatype: *none*)

Synonyms for *TINYINT*.

BLOB (SQL2003 Datatype: *BLOB*)

Stores up to 65,535 characters of data. Support for indexing *BLOB* columns is found only in MySQL Version 3.23.2 or greater (a feature not found in any other platform covered in this book). In MySQL, *BLOBs* are functionally equivalent to the MySQL datatype *VARCHAR BINARY* (discussed below) with the default upper limit. *BLOBs* always require case-sensitive comparisons. They differ from a MySQL *VARCHAR BINARY* column by not allowing *DEFAULT* values and not removing trailing spaces. Do not perform *GROUP BY* or *ORDER BY* on *BLOB* columns. They also are stored separately from their table, whereas all other datatypes in MySQL are stored in the table file structure itself.

CHAR(n)[BINARY], *CHARACTER(n) [BINARY]* (SQL2003 Datatype: *CHARACTER(n)*)

Contains a fixed-length character string of 1 to 255 characters in length. *CHAR* pads with blank spaces when it

stores values, but trims spaces upon retrieval as ANSI SQL2003 *VARCHAR* does. The *BINARY* option allows binary searches rather than dictionary-order, case-insensitive searches.

DATE (SQL2003 Datatype: DATE)

Stores a date from 1000-01-01 to 9999-12-31 (delimited by quotes). MySQL displays these values by default as YYYY-MM-DD, though the user may specify some other display format.

DATETIME (SQL2003 Datatype: TIMESTAMP)

Stores date and time values within the range of 1000-01-01 00:00:00 to 9999-12-31 23:59:59.

DECIMAL [(p[,s])] [ZEROFILL] (SQL2003 Datatype: DECIMAL(PRECISION, SCALE))

Stores exact numeric values as if they were strings, using a single character for each digit. Precision is 10 if omitted, and scale is 0 if omitted.

DOUBLE[(p,s)] [ZEROFILL], DOUBLE PRECISION[(p,s)] [ZEROFILL] (SQL2003 Datatype: DOUBLE PRECISION)

Holds double-precision numeric values and is otherwise identical to the double-precision *FLOAT* datatype, except that its allowable range is -1.7976931348623157E+308 to -2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308.

ENUM ("val1""val2," . . . n) (SQL2003 Datatype: none)

A datatype whose value must be one of those contained in the list of values (expressed as strings but stored as integers), NULL, or an empty string("") as an error value. Up to 65,535 distinct values are allowed.

FLOAT[(p)] [ZEROFILL], float(p,s) [ZEROFILL] (SQL2003 Datatype: FLOAT(P))

Stores floating-point numbers in the range -3.402823466E+38 to -1.175494351E-38 and 1.175494351E-38 to 3.402823466E+38. *FLOAT* without a precision, or with a precision of ≤ 24 , is single precision. Otherwise, *FLOAT* is double precision. When specified alone, precision can range from 0 to 53. When you specify both precision and scale, precision may be as high as 255 and the scale may be as high as 253. All *FLOAT* calculations in MySQL are done with double precision and may, since *FLOAT* is an approximate datatype, encounter rounding errors.

INT[EGER] [(n)] [UNSIGNED] [ZEROFILL] [AUTO_INCREMENT] (SQL2003 Datatype: INT, INTEGER)

Stores signed or unsigned integers within the range of -2,147,483,648 to 2,147,483,647 on ISAM tables. Unsigned, the range is 0 to 4,294,967,295 on ISAM tables. The range of values varies slightly on other types of tables. *AUTO_INCREMENT* is available to all of the *INT* variants. It creates a unique row identity for all new rows added to the table. (Refer to [CREATE/ALTER TABLE Statement](#) for more information on *AUTO_INCREMENT*.)

LONGBLOB (SQL2003 Datatype: BINARY LARGE OBJECT)

Stores *BLOB* data up to 4,294,967,295 characters in length. Note that this might be too much information for some client/server protocols to support.

LONGTEXT (SQL2003 Datatype: CLOB)

Stores *TEXT* data up to 4,294,967,295 characters in length. Note that this might be too much data for some client/server protocols to support.

MEDIUMBLOB (SQL2003 Datatype: none)

Stores *BLOB* data up to 16,777,215 characters in length.

MEDIUMTEXT (SQL2003 Datatype: none)

Stores *TEXT* data up to 16,777,215 characters in length.

MEDIUMINT[(*n*)] [*UNSIGNED*] [*ZEROFILL*] (*SQL2003 Datatype: none*)

Stores signed or unsigned integers within the range of 8,388,608 to -8,388,608. The unsigned range is 0 to 16,777,215.

NCHAR(*n*) [*BINARY*], [*NATIONAL*] *CHAR*(*n*) [*BINARY*] (*SQL2003 Datatype: NCHAR(n)*)

Synonyms for *CHAR*. The *NCHAR* datatypes provide UNICODE support beginning in MySQL v4.1.

NUMERIC(*p,s*) (*SQL2003 Datatype: DECIMAL(p,s)*)

A synonym of *DECIMAL*.

NVARCHAR(*n*) [*BINARY*], [*national*] *VARCHAR*(*n*) [*BINARY*], *NATIONAL CHARACTER VARYING*(*n*) [*BINARY*] (*SQL2003 Datatype: NCHAR VARYING*)

Synonyms for *VARYING [BINARY]*. Holds variable length character strings up to 255 characters in length. Values are stored and compared as case-insensitive unless the *BINARY* keyword is used.

REAL(*p,s*) (*SQL2003 Datatype: REAL*)

Is a synonym of *DOUBLE PRECISION*.

SET("val1," "val2," . . . *n*) (*SQL2003 Datatype: none*)

Is a *CHAR* datatype whose value must be equal to zero or more values specified in the list of values. Up to 64 items are allowed in the list of values.

SMALLINT[(*n*)] [*UNSIGNED*] [*ZEROFILL*] (*SQL2003 Datatype: SMALLINT*)

Stores signed or unsigned integers. The signed range is from -32768 to 32,767. The unsigned range is 0 to 65,535.

TEXT (*SQL2003 Datatype: none*)

Stores up to 65,535 characters of data. *TEXT* datatypes are stored separately from their tables, whereas all other datatypes are stored in their respective table file structures. *TEXT* is functionally equivalent to *VARCHAR* with no specific upper limit (besides the maximum size of the column), and requires case-insensitive comparisons. *TEXT* differs from a standard *VARCHAR* column by not allowing *DEFAULT* values, and by not removing trailing spaces. *TEXT* columns cannot be used in *GROUP BY* or *ORDER BY* clauses. In addition, support for indexing *TEXT* columns comes only in MySQL Version 3.23.2 or greater.

2.3.3 Oracle Datatypes

As shown below, Oracle supports a rich variety of datatypes, including most of the SQL2003 datatypes.

BFILE (*SQL2003 Datatype: DATALINK*)

Holds a pointer to a *BLOB* stored outside the database, but present on the local server, of up to 4 GB in size. The database streams input (but not output) access to the external *BLOB*. If you delete a row containing a *BFILE* value, only the pointer value is deleted. The actual file structure is not deleted.

BINARY_FLOAT (*SQL2003 Datatype: FLOAT*)

Holds a 32-bit floating point number.

BINARY_DOUBLE (*SQL2003 Datatype: FLOAT*)

Holds a 64-bit floating point number.

BLOB (SQL2003 Datatype: *BLOB*)

Holds a binary large object (*BLOB*) value of between 8 and 128 terabytes in size, depending on the database blocksize. In Oracle, large binary objects (*BLOBs*, *CLOBs*, and *NLOBs*) have the following restrictions:

- They cannot be selected remotely.
- They cannot be stored in clusters.
- They cannot compose a varray.
- They cannot be a component of an *ORDER BY* or *GROUP BY* clause in a query.
- They cannot be used by an aggregate function in a query.
- They cannot be referenced in queries using *DISTINCT*, *UNIQUE*, or joins.
- They cannot be referenced in *ANALYZE...COMPUTE* or *ANALYZE...ESTIMATE* statements.
- They cannot be part of a primary key or index key.
- They cannot be used in the *UPDATE OF* clause in an *UPDATE* trigger.

CHAR(n)[BYTE | CHAR], *CHARACTER(n)[BYTE | CHAR]* (SQL2003 Datatype: *CHARACTER(n)*)

Holds fixed-length character data up to 2,000 bytes in length. *BYTE* tells Oracle to use bytes for the size measurement. *CHAR* tells Oracle to use characters for the size measurement.

CLOB (SQL2003 Datatype: *CLOB*)

Stores a character large object (*CLOB*) value of between 8 and 128 terabytes in size, depending on the database blocksize. See *BLOB* for a list of restrictions on the use of the *CLOB* type.

DATE (SQL2003 Datatype: *DATE*)

Stores a valid date and time within the range of 4712BC-01-01 00:00:00 to 9999AD-12-31 23:59:59.

DECIMAL(p,s) (SQL2003 Datatype: *DECIMAL(p,s)*)

Is a synonym of *NUMBER* that accepts precision and scale arguments.

DOUBLE PRECISION (SQL2003 Datatype: *DOUBLE PRECISION*)

Stores floating point values with double precision, the same as *FLOAT(126)*.

FLOAT(n) (SQL2003 Datatype: *FLOAT(n)*)

Stores floating-point numeric values with a binary precision of up to 126.

INTEGER(n) (SQL2003 Datatype: *INTEGER*)

Stores signed and unsigned integer values with a precision of up to 38. *INTEGER* is treated as a synonym for *NUMBER*.

INTERVAL DAY (n) TO SECOND (x) (SQL2003 Datatype: *INTERVAL*)

Stores a time span in days, hours, minutes, and seconds where *n* is the number of digits in the day field (0 to 9 acceptable, 2 is the default) and *x* is the number of digits used for fractional seconds in the seconds field (0 to 9 acceptable, 6 is the default).

INTERVAL YEAR (n) TO MONTH (SQL2003 Datatype: *INTERVAL*)

Stores a time span in years and months where *n* is the number of digits in the year field. The value *n* can be 0

to 9, with a default of 2.

LONG (SQL2003 Datatype: none)

Stores variable-length character data up to 2 gigabytes in size. Please note, however, that *LONG* is not scheduled for long-term support by Oracle. Use another datatype, such as *CLOB*, instead of *LONG*, whenever possible.

LONG RAW (SQL2003 Datatype: none)

Stores raw variable-length binary data up to 2 gigabytes in size. *LONG RAW* and *RAW* are typically used to store graphics, sounds, documents, and other large data structures. *BLOB* is preferred over *LONGRAW* in Oracle because *BLOB* has fewer restrictions on its use. *LONGRAW* is also deprecated.

NATIONAL CHARACTER VARYING (n), *NATIONAL CHAR VARYING (n)*,

NCHAR VARYING (n) (SQL2003 Datatype: *NCHAR VARYING (n)*)

The same as *NVARCHAR2*.

NCHAR(n), *NATIONAL CHARACTER(n)*, *NATIONAL CHAR(n)* (SQL2003 Datatype: *NATIONAL CHARACTER*)

Holds UNICODE character data of 1 to 2,000 bytes in length. Default size is 1 byte.

NCLOB (SQL2003 Datatype: *NCLOB*)

Represents a *CLOB* that supports multibyte and UNICODE values of between 8 and 128 terabytes in size, depending on the database blocksize. See *BLOB* for a list of restrictions on the use of the *CLOB* type.

NUMBER (p,s), *NUMERIC (p,s)* (SQL2003 Datatype: *NUMERIC (p,s)*)

Stores a number with a precision of 1 to 38 and a scale of -84 to 127.

NVARCHAR2(n) (SQL2003 Datatype: none)

Represents Oracle's preferred UNICODE variable-length character datatype. It can hold 1 to 4,000 bytes.

RAW(n) (SQL2003 Datatype: none)

Stores raw, variable-length binary data up to 2,000 bytes in size. The value *n* is the specified size of the datatype. *RAW* is also deprecated in Oracle 10g. See *LONG RAW*.

REAL (SQL2003 Datatype: *REAL*)

Stores floating-point values as single-precision and is the same as *FLOAT(63)*.

ROWID (SQL2003 Datatype: none)

Represents a unique, base-64 identifier for each row in a table, often used in conjunction with the *ROWID* pseudocolumn.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

The same as *INTEGER*.

TIMESTAMP(n){[WITH TIME ZONE] | [WITH LOCAL TIME ZONE]} (SQL2003 Datatype: *TIMESTAMP[WITH TIME ZONE]*)

A full date and time value where *n* is the number of digits (0 to 9 are acceptable, 6 is the default) in the fractional part of the seconds field. *WITH TIME ZONE* stores whatever time zone you pass to it (the default is your session time zone) and returns a time value in that same time zone. *WITH LOCAL TIME ZONE* stores data in the time zone of the current session and returns data in the time zone of the user's session.

UROWID [(n)] (SQL2003 Datatype: none)

Stores a base-64 value showing the logical address of the row in its table. It defaults to 4,000 bytes in size, but you may optionally specify its size up to 4,000 bytes.

VARCHAR(n), *CHARACTER VARYING(n)*, *CHAR VARYING(n)* (SQL2003 Datatype: *CHARACTER VARYING(n)*)

Holds variable-length character data of 1 to 4,000 bytes in size.



Oracle does not recommend using *VARCHAR*. Oracle has encouraged the use of *VARCHAR2* instead for many years.

VARCHAR2(n [BYTE | CHAR]) (SQL2003 Datatype: *CHARACTER VARYING(n)*)

Holds variable-length character data up to 4,000 bytes in length as defined by *n*. *BYTE* tells Oracle to use bytes for the size measurement. *CHAR* tells Oracle to use characters for the size measurement. If you use *CHAR*, Oracle internally must still transform that into some number of bytes, which is then subject to the 4,000-byte upper limit.

XMLTYPE (SQL2003 Datatype: *XML*)

Stores XML data within the Oracle database. The XML data is accessed using XPath expressions as well as a number of built-in XPath functions, SQL functions, and PL/SQL packages. The *XMLTYPE* datatype is a system-defined type, so it is usable as an argument in functions, or as a datatype of column in a table or view. When used in a table, the data can be stored in a *CLOB* column or object-relationally.

2.3.4 PostgreSQL Datatypes

PostgreSQL database supports most SQL2003 datatypes, plus an extremely rich set of datatypes that store spatial and geometric data. PostgreSQL sports a rich set of operators and functions especially for the geometric datatypes, including capabilities such as rotation, finding intersections, and scaling. PostgreSQL also supports additional versions of existing datatypes that are smaller and take up less disk space than their corresponding primary datatypes. For example, PostgreSQL offers several variations on *INTEGER* to accommodate small or large numbers and thereby consume proportionally less or more space.

BIGSERIAL

See *SERIALS*.

BIT (SQL2003 Datatype: *BIT*)

A fixed-length bit string.

BIT VARYING(n), *varbit(n)* (SQL2003 Datatype: *BIT VARYING*)

Denotes a variable-length bit string *n* bits in length.

BOOL, *BOOLEAN* (SQL2003 Datatype: *BOOLEAN*)

Stores a logical Boolean (true/false/unknown) value. The keywords *TRUE* and *FALSE* are preferred, but PostgreSQL supports the following valid literal values for the "true" state: *TRUE*, *t*, *true*, *y*, *yes*, and *1*. Valid "false" values are: *FALSE*, *f*, *false*, *n*, *no*, and *0*.

BOX (*x1*, *y1*), (*x2*, *y2*)) (SQL2003 Datatype: none)

Stores the values of a rectangular box in a 2D plane. Values are stored in 32 bytes and are represented as ((*x1*,*y1*),(*x2*,*y2*)), signifying the opposite corners of the box (upper-right and lower-left corners, respectively). The outer parentheses are optional.

BYTEA (SQL2003 Datatype: *BINARY LARGE OBJECT*)

Raw, binary data such as that used to store graphics, sound, or documents. For storage, this datatype requires 4 bytes plus the actual size of the binary string.

CHAR(*n*), *CHARACTER*(*n*) (SQL2003 Datatype: *CHARACTER*(*n*))

Contains a fixed-length character string padded with spaces up to a length of *n*. Attempting to insert a value longer than *n* results in an error (unless the extra length is composed of spaces, which are then truncated such that the result fits in *n* characters).

CIDR (*x.x.x.x/y*) (SQL2003 Datatype: none)

Describes an IP-Version 4 network or host address in a 12-byte storage space. The range is any valid IPv4 network address. Data in *CIDR* datatypes is represented as *x.x.x.x/y* where the *x*s are the IP address and *y* is the number of bits in the netmask. *CIDR* does not accept nonzero bits to the right of a zero bit in the netmask.

CIRCLE *x*, *y*, *r* (SQL2003 Datatype: none)

Describes a circle in a 2D plane. Values are stored in 24 bytes of storage space and are represented as: *x*, *y*, *r*. The *x*, *y* value represents the coordinates of the center of the circle, while *r* represents the length of the radius. Parentheses or arrow brackets may optionally delimit the values for *x*, *y*, and *r*.

DATE (SQL2003 Datatype: *DATE*)

Holds a calendar date (year, day, and month) without the time of day in a 4-byte storage space. Dates must be between 4713 BC and 32767 AD. *DATE*'s lowest resolution, naturally, is to the day.

DATETIME (SQL2003 Datatype: *TIMESTAMP*)

Holds a calendar date with a specific time of day.

DECIMAL [(*p*,*s*)], *NUMERIC*[(*p*,*s*)](SQL2003 Datatype: *DECIMAL* (*PRECISION*, *SCALE*), *NUMERIC* (*p*,*s*))

Stores exact numeric values with a precision (*p*) as high as 9 and a scale (*s*) of 0, with no upper limit.

FLOAT4, *REAL* (SQL2003 Datatype: *FLOAT*(*p*))

Stores floating-point numbers with a precision of 8 or less and 6 decimal places.

FLOAT8, *DOUBLE PRECISION* (SQL2003 Datatype: *FLOAT*(*p*), 7 <= *p* < 16)

Stores floating-point numbers with a precision of 16 or less, and 15 decimal places.

INET (*x.x.x.x/y*)

Stores an IP-Version 4 network or host address in a 12-byte storage space. The range is any valid IPv4 network address. The *x*'s represent the IP address, and *y* is the number of bits in the netmask. The netmask defaults to 32. Unlike *CIDR*, *INET* accepts nonzero bits to the right of the netmask.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

Stores signed or unsigned 2-byte integers within a range of -32,768 to 32,767. *INT2* is a synonym.

INTEGER (SQL2003 Datatype: *INTEGER*)

Stores signed or unsigned 4-byte integers within a range of -2,147,483,648 to 2,147,483,647. *INT4* is a synonym.

INT8 (SQL2003 Datatype: none)

Stores signed or unsigned 8-byte integers with a range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

INTERVAL(p) (SQL2003 Datatype: none)

Holds general-use time-span values within the range of -178,000,000 to 178,000,000 years in a 12-byte storage space. *INTERVAL*'s lowest resolution is to the microsecond. This is a different datatype than the ANSI standard, which requires an interval qualifier such as *INTERVAL YEAR TO MONTH*.

LINE ((*x1*, *y1*), (*x2*, *y2*)) (SQL2003 Datatype: none)

Holds line data, without endpoints, in 2D plane values. Values are stored in 32 bytes and are represented as ((*x1*,*y1*), (*x2*,*y2*)), indicating the start and end points of a line. Parentheses are optional for line syntax.

LSEG ((*x1*, *y1*), (*x2*, *y2*)) (SQL2003 Datatype: none)

Holds line segment (*LSEG*) data, with endpoints, in a 2D plane. Values are stored in 32 bytes and are represented as ((*x1*,*y1*), (*x2*,*y2*)). Parentheses are optional for *LSEG* syntax. To those who are interested, the "line segment" is what most people traditionally think of as a line. For example, the lines on a playing field are actually line segments.



In true geometric nomenclature, a *line* stretches to infinity, having no terminus at either end, while a *line segment* has end points. PostgreSQL allows datatypes for both, but they are functionally equivalent.

MACADDR (SQL2003 Datatype: none)

Holds a value for the *MAC* address of a computer's network interface card in a 6-byte storage space. *MACADDR* accepts a number of industry standard representations, such as:

08002B:010203
08002B-010203
0800.2B01.0203
08-00-2B-01-02-03
08:00:2B:01:02:03

MONEY, *DECIMAL(9,2)* (SQL2003 Datatype: none)

Stores U.S.-style currency values in the range of (-21,474,836.48 to 21,474,836.47).

NUMERIC [(*p*, *s*)], *DECIMAL* [(*p*, *s*)] (SQL2003 Datatype: none)

Stores exact numeric values with a precision (*p*) and scale (*s*).

OID (SQL2003 Datatype: none)

Stores unique object identifiers.

PATH ((*x1*, *y1*), ...*n*), *Path* [(*x1*, *y1*), ...*n*] (SQL2003 Datatype: none)

Describes an open and closed geometric path in a 2D plane. Values are represented as $[(x1, y1), \dots]$ and consume $4 + 32n$ bytes of storage space. Each (x,y) value represents a point on the path. Paths are either open, where the first and last points do not intersect, or closed, where the first and last points do intersect. Parentheses are used to encapsulate closed paths, while brackets encapsulate open paths.

POINT (x, y) (SQL2003 Datatype: none)

Stores values for a geometric point in a 2D plane in a 16-byte storage space. Values are represented as (x,y) . The point is the basis for all other two-dimensional spatial datatypes supported in PostgreSQL. Parentheses are optional for point syntax.

POLYGON ($(x1, y1), \dots, n$) (SQL2003 Datatype: none)

Stores values for a closed geometric path in a 2D plane using $4 + 32n$ bytes of storage. Values are represented as $[(x1,y1), \dots]$. *POLYGON* is essentially a closed path datatype.

SERIAL, *SERIAL4* (SQL2003 Datatype: none)

Stores an autoincrementing, unique, integer ID for indexing and cross-referencing. These types store up to 4 bytes of data (a range of numbers from 1 to 2,147,483,647). Tables defined with this datatype cannot be directly dropped. You must first issue the *DROP SEQUENCE* command, then follow up with the *DROP TABLE* command.

SERIAL8, *BIGSERIAL* (SQL2003 Datatype: none)

Stores an autoincrementing, unique integer ID for indexing and cross-referencing. It stores up to 8 bytes of data (a range of numbers from 1 to 9,223,372,036,854,775,807). Tables defined with this datatype cannot be directly dropped. You must first issue the *DROP SEQUENCE* command, and then follow up with the *DROP TABLE* command.

TEXT (SQL2003 Datatype: *CLOB*)

Stores large, variable-length, character-string data up to 1 gigabyte. PostgreSQL automatically compresses *TEXT* strings, so the disk size may be less than the string size.

TIME [(p)] [*WITHOUT TIME ZONE* | *WITH TIME ZONE*] (SQL2003 Datatype: *TIME*)

Holds the time of day and stores either no time zone (using 8 bytes of storage space) or the time zone of the database server (using 12 bytes of storage space). The allowable range is from 00:00:00.00 to 23:59:59.99. The lowest granularity is 1 microsecond. Note that time zone information on most Unix systems is available only for the years 1902 through 2038.

TIMESPAN (SQL2003 Datatype: none)

Holds a value that represents a specific span of time. The ANSI datatype most like PostgreSQL's *TIMESPAN* is *INTERVAL*.

TIMESTAMP [(p)] [*WITHOUT TIME ZONE* | *WITH TIME ZONE*] (SQL2003 Datatype: *TIMESTAMP* [*WITH TIME ZONE* | *WITHOUT TIME ZONE*])

Stores the date and time and stores either no time zone or the time zone of the database server. The range of values is from 4713 BC to 1465001 AD. *TIMESTAMP* uses 8 bytes of storage space per value. The lowest granularity is 1 microsecond. Note that time zone information on most Unix systems is available only for the years 1902 through 2038.

TIMETZ (SQL2003 Datatype: *TIME WITH TIME ZONE*)

Holds the time of day, including the time zone.

VARCHAR(n), *CHARACTER VARYING*(n) (SQL2003 Datatype: *CHARACTER VARYING*(n))

Stores variable-length character strings up to a length of n . Trailing spaces are not stored.

2.3.5 SQL Server Datatypes

The list below shows that Microsoft SQL Server supports most SQL2003 datatypes. SQL Server supports additional datatypes used to uniquely identify rows of data within a table and across multiple servers, such as *UNIQUEIDENTIFIER*, in support of Microsoft's hardware philosophy of "scale-out" (that is, deploying on many Intel-based servers), rather than "scale-up" (deploying on a single, huge, high-end Unix server or a Windows Data Center Server).



An interesting side note about SQL Server dates: SQL Server supports dates starting at the year 1753. You can't store dates prior to that year using any of SQL Server's date datatypes. Why not? The rationale is that the English-speaking world started using the Gregorian calendar in 1753 (the Julian calendar was used prior to September, 1753) and converting dates prior to Julian to the Gregorian calendar can be quite challenging.

BIGINT (SQL2003 Datatype: *BIGINT*)

Stores signed and unsigned integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 using 8 bytes of storage space. See *INT* for *IDENTITY* property rules that also apply to *BIGINT*.

BINARY[(*n*)] (SQL2003 Datatype: *BLOB*)

Stores a fixed-length binary value of 1 to 8,000 bytes in size. *BINARY* datatypes consume $n + 4$ bytes of storage space.

BIT (SQL2003 Datatype: *BOOLEAN*)

Stores 1, 0, or NULL to indicate "unknown." Up to 8 *BIT* columns on a single table will be stored in a single byte. An additional 8 *BIT* columns consume one more byte of storage space. *BIT* columns cannot be indexed.

CHAR[(*n*)], *CHARACTER*[(*n*)] (SQL2003 Datatype: *CHARACTER*(*n*))

Holds fixed-length character data of 1 to 8,000 characters in length. Any unused space is, by default, padded with spaces. (You can disable the automatic padding of spaces.) Storage size is n bytes.

CURSOR (SQL2003 Datatype: none)

A special datatype used to describe a cursor as a variable or stored procedure *OUTPUT* parameter. It cannot be used in a *CREATE TABLE* statement. The *CURSOR* datatype is always nullable.

DATETIME (SQL2003 Datatype: *TIMESTAMP*)

Holds date and time data within the range of 1753-01-01 00:00:00 through 9999-12-31 23:59:59. Values are stored in an 8-byte storage space.

DECIMAL (*p,s*), *DEC* (*p,s*), *NUMERIC* (*p,s*) (SQL2003 Datatype: *DECIMAL* (*p,s*), *NUMERIC* (*p,s*))

Stores decimal values up to 38 digits long. The values p and s define precision and scale, respectively. The default value for scale is 0. The precision of the datatype determines how much storage space it will consume:

- Precision 1-9 uses 5 bytes
- Precision 10-19 uses 9 bytes
- Precision 20-28 uses 13 bytes
- Precision 29-39 uses 17 bytes

See *INT* for *IDENTITY* property rules that also apply to *DECIMAL*.

DOUBLE PRECISION (SQL2003 Datatype: none)

A synonym for *FLOAT*(53).

FLOAT [(*n*)] (SQL2003 Datatype: *FLOAT*, *FLOAT* (*n*))

Holds floating-point numbers in the range -1.79E+308 through 1.79E+308. Precision, represented by *n*, may be 1 to 53. Storage size is 4 bytes for 7 digits, where *n* is 1 to 24. Anything larger requires 8 bytes of storage.

IMAGE (SQL2003 Datatype: *BLOB*)

Stores a variable-length binary value up to 2,147,483,647 bytes in length. This datatype is commonly used to store graphics, sounds, and files like MS-Word documents and MS-Excel spreadsheets. *IMAGE* cannot be freely manipulated. *IMAGE* and *TEXT* columns have a lot of constraints on how they can be used. See *TEXT* for a list of the commands and functions that work on an *IMAGE* datatype.

INT [*IDENTITY* [(*seed*, *increment*)]] (SQL2003 Datatype: *INTEGER*)

Stores signed or unsigned integers between -2,147,483,648 and 2,147,483,647 in 4 bytes of storage space. All integer datatypes, as well as the decimal type, support the *IDENTITY* property. An identity is an automatically incrementing row identifier. Refer to the [CREATE/ALTER TABLE Statement](#).

MONEY (SQL2003 Datatype: none)

Stores monetary values in a range from -922,337,203,685,477.5808 to 922,337,203,685,477.5807. Values are stored in an 8-byte storage space.

NCHAR(*n*), *NATIONAL CHAR*(*n*), *NATIONAL CHARACTER*(*n*) (SQL2003 Datatype: *NATIONAL CHARACTER* (*n*))

Holds fixed-length UNICODE data up to 4,000 characters in length. *NCHAR* consumes twice *n* in storage space.

NTEXT, *NATIONAL TEXT* (SQL2003 Datatype: *NCLOB*)

Holds UNICODE text passages up to 1,073,741,823 characters in length. See *TEXT* for rules about the commands and functions available for *NTEXT*.

NUMERIC(*p,s*) (SQL2003 Datatype: *DECIMAL*(*p,s*))

A synonym for *DECIMAL*. See *INT* for rules about the *IDENTITY* property.

NVARCHAR(*n*), *NATIONAL CHAR VARYING*(*n*), *NATIONAL CHARACTER VARYING*(*n*) (SQL2003 Datatype: *NATIONAL CHARACTER VARYING*(*n*))

Holds variable-length UNICODE data up to 4,000 characters in length. Storage space consumed is double the character length (characters * 2) inserted into the field. The system setting *SET ANSI_PADDING* is always enabled (ON) for *NCHAR* and *NVARCHAR* fields in SQL Server.

REAL, *FLOAT*(24) (SQL2003 Datatype: *REAL*)

Holds floating point numbers of -3.40E + 38 through 3.40 E+38 in a 4-byte storage space. *REAL* is functionally equivalent to *FLOAT*(24).

ROWVERSION (SQL2003 Datatype: none)

A unique number within a database that is updated whenever a row is updated. Called *TIMESTAMP* in earlier versions.

SMALLDATETIME (SQL2003 Datatype: none)

Holds date and time data within the range of `1900-01-01 00:00' through `2079-06-06 23:59' down to the nearest minute. (Minutes are rounded down when seconds are 29.998 or less. Otherwise, they are rounded up.) Values are stored in a 4-byte storage space.

SMALLINT (SQL2003 Datatype: *SMALLINT*)

Stores signed or unsigned integers between -32,768 and 32,767 in 2 bytes of storage space. See *INT* for rules about the *IDENTITY* property, which applies to this type.

SMALLMONEY (SQL2003 Datatype: none)

Stores monetary values within the range of 214,748.3648 and -214,748.3647. Values are stored in a 4-byte storage space.

SQL_VARIANT (SQL2003 Datatype: none)

Stores values of other SQL Server-supported datatypes, except *TEXT*, *NTEXT*, *ROWVERSION*, and other *SQL_VARIANT*s. It can store up to 8,016 bytes of data and supports NULL and DEFAULT values. *SQL_VARIANT* is used in columns, parameters, variables, and return values of functions and stored procedures.

TABLE (SQL2003 Datatype: none)

A special datatype that stores a result set for a later process. It is used solely in procedural processing and *cannot* be used in a *CREATE TABLE* statement. This datatype alleviates the need for temporary tables in many applications. It can reduce the need for procedure recompiles, thus speeding execution of stored procedures and user-defined functions.

TEXT (SQL2003 Datatype: *CLOB*)

Stores very large passages of text up to 2,147,483,647 characters in length. *TEXT* and *IMAGE* values are often more difficult to manipulate than, say, *VARCHAR* values. For example, you cannot place an index on a *TEXT* or *IMAGE* column. *TEXT* can be manipulated using the functions *DATALength*, *PATINDEX*, *SUBSTRING*, *TEXTPTR*, and *TEXTVALID* as well as the commands *READTEXT*, *SET TEXTSIZE*, *UPDATETEXT*, and *WRITETEXT*.

TIMESTAMP (SQL2003 Datatype: *TIMESTAMP*)

Stores an automatically generated binary number that guarantees uniqueness in the current database and is therefore different from the ANSI *TIMESTAMP* datatype. *TIMESTAMPS* consume 8 bytes of storage space. *ROWVERSION* is now preferred over *TIMESTAMP* to uniquely track each row.

TINYINT (SQL2003 Datatype: none)

Stores unsigned integers between 0 and 255 in 1 byte of storage space. See the entry for *INT* to see rules about the *IDENTITY* property, which applies to this type.

UNIQUEIDENTIFIER (SQL2003 Datatype: none)

Represents a value that is globally unique across all databases and all servers. Values are represented as *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, in which each *x* is a hexadecimal digit in the range 0-9 or a-f. The only operations allowed against *UNIQUEIDENTIFIERS* are comparisons and NULL checks. Column constraints and properties are allowed on *UNIQUEIDENTIFIER* columns, with the exception of the *IDENTITY* property.

VARBINARY[(*n*)] (SQL2003 Datatype: *BLOB*)

Describes a variable-length binary value up to 8,000 bytes in size. Storage space consumed is equivalent to the size of the data inserted, plus 4 bytes.

VARCHAR[(*n*)], *CHAR VARYING*[(*n*)], *CHARACTER VARYING* [(*n*)] (SQL2003 Datatype: *CHARACTER VARYING*(*n*))

Holds fixed-length character data of 1 to 8,000 characters in length. Storage space is the actual size of the value entered in bytes, not the value of *n*.

2.4 Constraints

Constraints allow you to automatically enforce the integrity of data and to filter the data that is placed in a database. In a sense, *constraints* are rules that define which data values are valid during *INSERT*, *UPDATE*, and *DELETE* operations. When a data modification transaction breaks the rules of a constraint, the transaction is rejected.

In the ANSI standard, there are four constraint types: *CHECK*, *PRIMARY KEY*, *UNIQUE*, and *FOREIGN KEY*. (The RDBMS platforms may allow more. Refer to different platform sections of the [CREATE/ALTER TABLE Statement](#) to find out these exceptions.)

2.4.1 Scope

Constraints may be applied at the column-level or the table-level:

Column-level constraints

Declared as part of a column definition and apply only to that column.

Table-level constraints

Declared independently from any column definitions (traditionally, at the end of a *CREATE TABLE* statement) and may apply to one or more columns in the table. A table constraint is required when you wish to define a constraint that applies to more than one column.

2.4.2 Syntax

Constraints are defined when you create or alter a table. The general syntax for constraints is shown here:

```
CONSTRAINT [constraint_name] constraint_type [(column [...])]  
[predicate] [constraint_deferment] [deferment_timing]
```

The syntax elements are as follows:

CONSTRAINT [*constraint_name*]

Begins a constraint definition and, optionally, provides a name for the constraint. When you omit *constraint_name*, the system will create one for you automatically. On some platforms, like DB2, you may omit the *CONSTRAINT* keyword as well. System-generated names are often incomprehensible. It is good practice to specify human-readable, sensible names for constraints.

constraint_type

Declares the constraint as one of the allowable types: *CHECK*, *PRIMARY KEY*, *UNIQUE*, or *FOREIGN KEY*. More information about each type of constraint appears later in this section.

column [...]

Associates one or more columns with the constraint. Specify the columns in a comma-delimited list. The column list must be enclosed in parentheses. The column list should be omitted for column-level constraints. Columns are not used in every constraint. For example, *CHECK* constraints do not generally use column references.

predicate

Defines a predicate for *CHECK* constraints.

constraint_deferment

Declares a constraint as *DEFERRABLE* or *NOT DEFERRABLE*. When a constraint is deferrable, you can specify that it be checked for a rules violation at the end of a transaction. When a constraint is not deferrable, it is checked for a rules violation at the conclusion of every SQL statement.

deferment_timing

Declares a deferrable constraint as *INITIALLY DEFERRED* or *INITIALLY IMMEDIATE*. When set to *INITIALLY DEFERRED*, the constraint check time will be deferred until the end of a transaction, even if it is composed of many SQL statements. In this case, the constraint must also be *DEFERRABLE*. When set to *INITIALLY IMMEDIATE*, the constraint check time is at the end of every SQL statement. In this case, the constraint may be either *DEFERRABLE* or *NOT DEFERRABLE*. The default is *INITIALLY IMMEDIATE*.

Note that the vendor platforms may have some variations on this syntax. Check different platform sections of [CREATE/ALTER TABLE Statement](#) for more details.

2.4.3 PRIMARY KEY Constraints

A *PRIMARY KEY* constraint declares one or more columns whose values uniquely identify each record in the table. It is considered a special case of the *UNIQUE* constraint. Some rules about primary keys:

- Only one primary key may exist on a table at a time.
- Columns in the primary key cannot contain datatypes of *BLOB*, *CLOB*, *NCLOB*, or *ARRAY*.
- Primary keys may be defined at the column level for a single column key or at the table level if multiple columns make up the primary key.
- Values in the primary key column(s) must be unique and not NULL.
- In a multicolumn primary key, called a *concatenated key*, the combination of values in all of the key columns must be unique and not NULL.
- Foreign keys can be declared that reference the primary key of a table to establish direct relationships between tables (or possibly, though rarely, within a single table).

The following ANSI standard code includes the options for creating both a table- and column-level primary key constraint on a table called **distributors**. The first example shows a column-level primary-key constraint, while the second shows a table-level constraint:

-- Creating a column-level constraint

```
CREATE TABLE distributors
```

```
  (dist_id   CHAR(4)  NOT NULL PRIMARY KEY,
```

```
   dist_name  VARCHAR(40),
```

```
   dist_address1 VARCHAR(40),
```

```
   dist_address2 VARCHAR(40),
```

```
   city       VARCHAR(20),
```

```
   state      CHAR(2)  ,
```

```
   zip        CHAR(5)  ,
```

```
   phone      CHAR(12) ,
```

```
   sales_rep  INT      );
```

-- Creating a table-level constraint

```
CREATE TABLE distributors
```

```
  (dist_id   CHAR(4)  NOT NULL,
```

```
   dist_name  VARCHAR(40),
```

```
dist_address1 VARCHAR(40),
dist_address2 VARCHAR(40),
city          VARCHAR(20),
state         CHAR(2)  ,
zip           CHAR(5)  ,
phone         CHAR(12) ,
sales_rep     INT      ,
```

CONSTRAINT pk_dist_id PRIMARY KEY (dist_id));

In the example showing a table-level primary key, we could easily have created a concatenated key by listing several columns separated by commas between each one.

2.4.4 FOREIGN KEY Constraints

A *FOREIGN KEY* constraint defines one or more columns in the table as referencing columns to a *UNIQUE* or *PRIMARY KEY* in another table. (A foreign key can reference a unique or primary key in the same table as the foreign key itself, but such foreign keys are rare.) Foreign keys can then prevent the entry of data into a table where there is no matching value in the related table. They are the primary means of identifying the relationship between tables in a relational database. Some rules about foreign keys:

- Many foreign keys may exist on a table at a time.
- A foreign key can be declared to reference either the primary key or a unique key of another table to establish a direct relationship between the two tables.

The full SQL2003 syntax for foreign keys is more elaborate than the general syntax for constraints shown earlier, and is dependent on whether you are making a table-level or column-level declaration:

-- Table-level foreign key

```
[CONSTRAINT [constraint_name] ]
FOREIGN KEY (local_column [,...])
REFERENCES referenced_table [ (referenced_column [,...]) ]
[MATCH {FULL | PARTIAL | SIMPLE} ]
[ON UPDATE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[ON DELETE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT} ]
[constraint_deferment] [deferment_timing]
```

-- Column-level foreign key

```
[CONSTRAINT [constraint_name] ]
REFERENCES referenced_table [ (referenced_column [,...]) ]
```


[MATCH {FULL | PARTIAL | SIMPLE}]

[ON UPDATE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT}]

[ON DELETE {NO ACTION | CASCADE | RESTRICT |
SET NULL | SET DEFAULT}]

[*constraint_deferment*] [*deferment_timing*]

Keywords common to a standard constraint declaration are described above, under the "Syntax" section. Keywords specific to foreign keys are described in the following list:

FOREIGN KEY (*local_column* [,...])

Declares one or more columns of the table being created or altered and are subject to the foreign key constraint. This syntax is used *only* in table-level declarations and is excluded from column-level declarations. We recommend that the ordinal position and datatype of each column in the *local_column* list match comparably with the ordinal position and datatype of the columns in the **referenced_column** list.

REFERENCES *referenced_table* [(*referenced_column* [,...])]

Names the table and, where appropriate, the column(s) that hold the valid list of values for the foreign key. A **referenced_column** must already be named in a *NOT DEFERRABLE PRIMARY KEY* or *NOT DEFERRABLE UNIQUE KEY*. The table types must match. For example, if one is a local temporary table then both must be local temporary tables.

MATCH {FULL | PARTIAL | SIMPLE}

Defines the degree of matching required between the local and referenced columns in foreign key constraints when NULLs are present.

FULL

Declares that a match is acceptable when: 1) none of the referencing columns are NULL and match all of the values of the referenced column, or 2) all of the referencing columns are NULL. In general, you should use *MATCH FULL* or else ensure that all columns involved have NOT NULL constraints.

PARTIAL

Declares that a match is acceptable when at least one of the referenced column is NULL and the others match the corresponding referenced columns.

SIMPLE

Declares that a match is acceptable when any of the values of the referencing column are NULL or a match. This is the default.

ON UPDATE

Specifies that, when an *UPDATE* operation affects one or more **referenced_columns** of the primary or unique key on the referenced table, a corresponding action should be taken to ensure the foreign key does not lose data integrity. *ON UPDATE* may be declared independently of or with the *ON DELETE* clause. When omitted, the default is *ON UPDATE NO ACTION*.

ON DELETE

Specifies that when a *DELETE* operation affects one or more **referenced_columns** of the primary or unique key on the referenced table, that a corresponding action should be taken to ensure the foreign key does not lose data integrity. *ON DELETE* may be declared independent of or with the *ON UPDATE* clause. When omitted, the default for the ANSI standard is *ON DELETE NO ACTION*.

NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT

Defines the action the database takes to maintain data integrity of the foreign key when a referenced primary or unique key constraint value is changed or deleted.

NO ACTION

Tells the database to do nothing when a primary key or unique key value, referenced by a foreign key, is changed or deleted.

CASCADE

Tells the database to perform the same action (i.e., *DELETE* or *UPDATE*) on the matching foreign key when a primary key or unique key value is changed or deleted.

RESTRICT

Tells the database to prevent changes to the primary key or unique key value, referenced by the foreign key.

SET NULL

Tells the database to set the value in the foreign key to NULL when a primary key or unique key value is changed or deleted.

SET DEFAULT

Tells the database to set the value in the foreign key to the default (using default values you specify for each column) when a primary key or unique key value is changed or deleted.

As with the code example for primary keys, you can adapt this generic syntax to both column-level and table-level foreign key constraints. Note that column-level and table-level constraints perform their function in exactly the same way. They are merely defined at different levels of the *CREATE TABLE* command. In the following example, we create a single-column foreign key on the **salesrep** column referencing the **empid** column of the **employee** table. We create the foreign key two different ways, the first time at the column level, the second time at the table level:

-- Creating a column-level constraint

CREATE TABLE distributors

```
(dist_id    CHAR(4)  PRIMARY KEY,  
dist_name  VARCHAR(40),  
dist_address1 VARCHAR(40),  
dist_address2 VARCHAR(40),  
city       VARCHAR(20),  
state      CHAR(2)  ,  
zip        CHAR(5)  ,  
phone      CHAR(12) ,  
sales_rep  INT       NOT  
NULL REFERENCES employee(empid));
```

```
-- Creating a table-level constraint
CREATE TABLE distributors
  (dist_id   CHAR(4)  NOT NULL,
   dist_name VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city      VARCHAR(20),
   state     CHAR(2)  ,
   zip       CHAR(5)  ,
   phone     CHAR(12) ,
   sales_rep INT      ,
  CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
  CONSTRAINT fk_empid FOREIGN KEY (sales_rep)
    REFERENCES employee(empid));
```

2.4.5 UNIQUE Constraints

A *UNIQUE* constraint, sometimes called a *candidate key*, declares that the values in one column, or the combination of values in more than one column, must be unique. Rules concerning unique constraints include:

- Columns in a unique key cannot contain datatypes of *BLOB*, *CLOB*, *NCLOB*, or *ARRAY*.
- The column or columns in a unique key may not be identical to those in any other unique keys, or to any columns in the primary key of the table.
- A single NULL value, if the unique key allows NULL values, is allowed.
- SQL2003 allows you to substitute the column list, shown in the general syntax diagram for constraints, with the keyword (*VALUE*). *UNIQUE (VALUE)* indicates that all columns in the table are part of the unique key. The *VALUE* keyword also disallows any other unique or primary keys on the table.

In the following example, we limit the number of distributors we do business with to only one distributor per ZIP Code. We also allow one (and only one) "catch-all" distributor with a NULL ZIP Code. This functionality can be implemented easily using a *UNIQUE* constraint, either at the column or the table level:

```
-- Creating a column-level constraint
CREATE TABLE distributors
  (dist_id   CHAR(4)  PRIMARY KEY,
   dist_name VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city      VARCHAR(20),
   state     CHAR(2)  ,
   zip       CHAR(5)  UNIQUE ,
```

```
    phone    CHAR(12) ,
    sales_rep INT     NOT NULL
REFERENCES employee(empid);

-- Creating a table-level constraint
CREATE TABLE distributors
    (dist_id    CHAR(4) NOT NULL,
    dist_name  VARCHAR(40),
    dist_address1 VARCHAR(40),
    dist_address2 VARCHAR(40),
    city      VARCHAR(20),
    state     CHAR(2)  ,
    zip       CHAR(5)  ,
    phone     CHAR(12) ,
    sales_rep INT      ,
    CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
    CONSTRAINT fk_emp_id  FOREIGN KEY (sales_rep)
        REFERENCES employee(empid),
CONSTRAINT unq_zip    UNIQUE (zip) );
```

2.4.6 CHECK Constraints

CHECK constraints allow you to perform comparison operations to ensure that values match specific conditions that you set out. The syntax for a check constraint is very similar to the general syntax for constraints:

```
[CONSTRAINT] [constraint_name] CHECK (search_conditions)
[constraint_deferment] [deferment_timing]
```

Other elements of the constraint are introduced earlier in this section. The following element is unique to the *CHECK* constraint:

search_conditions

Specifies one or more search conditions that constrain the values inserted into the column or table, using one or more expressions and a predicate. Multiple search conditions may be applied to a column in a single check constraint using the *AND* and *OR* operators. (Think of a *WHERE* clause.)

A check constraint is considered matched when the search conditions evaluate to TRUE or UNKNOWN. Check constraints are limited to Boolean operations (e.g., =, >=, <=, or <>), though they may include any SQL2003 predicates such as *IN* or *LIKE*. Check constraints may be appended to one another (when checking a single column) using the *AND* and *OR* operators. Some other rules about *CHECK* constraints:

- A column or table may have one or more *CHECK* constraints.
- A search condition cannot contain aggregate functions, except in a subquery.
- A search condition cannot use non-deterministic functions or subqueries.

- A check constraint can only reference like objects. So if a check constraint is declared on a global temporary table, it cannot then reference a permanent table.
- A search condition cannot reference these ANSI functions: *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *CURRENT_TIMESTAMP*, *LOCALTIME*, and *LOCALTIMESTAMP*.

The following example adds a check constraint to the **dist_id** and **zip** columns. (This example uses generic code run on SQL Server.) The ZIP Code must fall into the normal ranges for postal ZIP Codes, while the **dist_id** values are allowed to contain either four alphabetic characters or two alphabetic and two numeric characters:

-- Creating column-level CHECK constraints

CREATE TABLE distributors

(dist_id CHAR(4)

CONSTRAINT pk_dist_id PRIMARY KEY

CONSTRAINT ck_dist_id CHECK

(dist_id LIKE '[A-Z][A-Z][A-Z][A-Z]' OR

dist_id LIKE '[A-Z][A-Z][0-9][0-9]'),

dist_name VARCHAR(40),

dist_address1 VARCHAR(40),

dist_address2 VARCHAR(40),

city VARCHAR(20),

state CHAR(2)

CONSTRAINT def_st DEFAULT ("CA"),

zip CHAR(5)

CONSTRAINT unq_dist_zip UNIQUE

CONSTRAINT ck_dist_zip CHECK

(zip LIKE '[0-9][0-9][0-9][0-9]'),

phone CHAR(12),

sales_rep INT

NOT NULL DEFAULT USER REFERENCES employee(emp_id))



Chapter 2. Foundational Concepts

SQL provides an easy, intuitive way to interact with a database. While the SQL2003 standard does not define the concept of a "database," it does define all the functions and concepts needed for a user to create, retrieve, update, and delete data. It is important to know the types of syntax in the SQL2003 standard and the particular platform-specific syntax guidelines.

3.1 How to Use This Chapter

When researching a command in this chapter:

1. Read [Section 3.2](#).
2. Check the platform support table.
3. Read the section on SQL2003 syntax and description even if you are looking for a specific platform implementation.
4. Finally, read the specific platform implementation information.

Any common features between the platform implementations of a command are discussed and compared against the SQL2003 topic. Thus, the subsection on a platform's implementation of a particular command may not describe every aspect of that command, since some of its details may be covered in the SQL2003 section. Please note that if there is a keyword that appears in a command syntax but not in its keyword description, this is because we chose not to repeat descriptions that appear under the ANSI entry.



If you skip straight to a platform implementation and do not see a discussion of a keyword or clause that you are interested in, that means it is described (in common across all supporting platforms) under the SQL2003 entry of the command.

3.2 SQL Platform Support

[Table 3-1](#) provides a listing of the SQL statements, the platforms that support them, and the degree to which they support them. The following list offers useful tips for reading [Table 3-1](#), as well as what each abbreviation stands for. The sections that follow describe the commands from the table in detail:

1. The first column contains the alphabetized SQL commands.
2. The SQL statement class for each command is indicated in the second column.
3. The subsequent columns list the vendor's level of support:

Supported (S)

The platform supports the SQL2003 standard for the particular command.

Supported, with variations (SWV)

The platform supports the SQL2003 standard for the particular command, using vendor-specific code or syntax.

Supported, with limitations (SWL)

The platform supports some but not all of the functions specified by the SQL2003 standard for the particular command.

Not supported (NS)

The platform does not support the particular command according to the SQL2003 standard.

Remember that even if a specific SQL2003 command is listed as "Not supported," the platform usually has alternative coding or syntax to enact the same command or function. Therefore, be sure to read the discussion and examples for each command later in this chapter. Likewise, a few of the commands in [Table 3-1](#) are not found in the SQL2003 standard, and these have been indicated with the term "Non-ANSI" under the heading "SQL2003 class" in the table.

Conversely, a few ANSI commands in the table (such as *CREATE DOMAIN* and *ALTER DOMAIN*) are not currently supported by any of the database platforms covered in this text. Since this book focuses on the implementation of the SQL language, unsupported ANSI commands are shown in [Table 3-1](#) but are not documented elsewhere in the book.

Table 3-1. Alphabetical quick SQL command reference

SQL2003 Command	SQL2003 class	DB2 v8.1	MySQL v4.0	Oracle v10g	Postgre SQL v7.2	SQL Server 2000
<i>ALL/ANY/SOME</i>	SQL-data	S	NS	S	S	S
<i>BETWEEN</i>	SQL-data	S	S	S	S	S
<i>ALTER DATABASE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>ALTER DOMAIN</i>	SQL-schema	NS	NS	NS	NS	NS
<i>ALTER FUNCTION</i>	SQL-schema	SWV	SWL	SWV	SWL	SWV
<i>ALTER INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>ALTER METHOD</i>	SQL-schema	SWV	NS	NS	NS	NS
<i>ALTER PROCEDURE</i>	SQL-schema	SWV	SWL	SWV	SWL	SWV
<i>ALTER TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV

<i>ALTER TRIGGER</i>	Non-ANSI	SWV	NS T	SWV	SWV	SWV
<i>ALTER TYPE</i>	SQL-schema	SWV	NS	SWV	NS	NS
<i>ALTER VIEW</i>	Non-ANSI	SWV	NS	SWV	SWV	SWV
<i>CALL</i>	SQL-control	SWV	NS T	SWV	NS	NS
<i>CLOSE CURSOR</i>	SQL-data	SWV	NS	S	S	SWV
<i>COMMIT</i>	SQL-transaction	S	SWV	SWV	SWV	SWV
<i>CONNECT</i>	SQL-connection	SWV	NS	S	NS	SWV
<i>CREATE DATABASE</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>CREATE DOMAIN</i>	SQL-schema	NS	NS	NS	NS	NS
<i>CREATE FUNCTION</i>	SQL-schema	SWV	SWL	SWV	SWL	SWV
<i>CREATE INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>CREATE METHOD</i>	SQL-schema	SWV	NS	NS	NS	NS
<i>CREATE PROCEDURE</i>	SQL-schema	S	NS	S	NS	S
<i>CREATE ROLE</i>	SQL-schema	NS	NS	SWV	NS	NS
<i>CREATE SCHEMA</i>	SQL-schema	SWL	NS	SWV	NS	SWL
<i>CREATE TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>CREATE TRIGGER</i>	SQL-schema	SWV	NS	SWV	SWV	SWV
<i>CREATE TYPE</i>	SQL-schema	SWV	NS	SWL	SWV	NS
<i>CREATE VIEW</i>	SQL-schema	SWV	NS	SWV	SWV	SWV
<i>DECLARE CURSOR</i>	SQL-data	SWL	NS	SWL	SWL	SWL
<i>DELETE</i>	SQL-data	SWV	SWV	SWV	S	SWL
<i>DISCONNECT</i>	SQL-connection	SWL	NS	SWL	NS	SWL
<i>DROP DATABASE</i>	Non-ANSI	NS	SWV	S	SWV	SWV
<i>DROP DOMAIN</i>	SQL-schema	NS	NS	NS	NS	NS
<i>DROP FUNCTION</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>DROP INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV	SWV
<i>DROP METHOD</i>	SQL-schema	SWV	NS	SWV	SWV	NS
<i>DROP PROCEDURE</i>	SQL-schema	SWV	NS	S	NS	S
<i>DROP ROLE</i>	SQL-schema	NS	NS	SWV	NS	NS
<i>DROP TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV

	Schema					
<i>DROP TYPE</i>	SQL-schema	S	NS	S	S	NS
<i>DROP TRIGGER</i>	SQL-schema	SWV	NS	SWV	SWV	SWV
<i>DROP VIEW</i>	SQL-schema	S	NS	S	S	S
<i>EXCEPT</i>	SQL-data	SWL	NS	SWL	SWL	NS
<i>EXISTS</i>	SQL-data	S	S	S	S	S
<i>FETCH</i>	SQL-data	SWL	NS	SWL	SWV	SWV
<i>GRANT</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>IN</i>	SQL-data	S	S	SWV	S	S
<i>INSERT</i>	SQL-data	SWV	SWV	SWV	S	SWV
<i>INTERSECT</i>	SQL-data	SWL	NS	SWL	SWL	NS
<i>IS</i>	SQL-data	S	S	S	S	S
<i>JOIN subclause</i>	SQL-data	SWL	SWV	S	SWV	SWL
<i>LIKE</i>	SQL-data	S	SWV	S	SWV	SWV
<i>MERGE</i>	SQL-data	S	NS	S	NS	NS
<i>OPEN</i>	SQL-data	SWV	NS	S	NS	S
<i>OPERATORS</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>ORDER BY</i>	SQL-data	SWL	SWL	SWV	SWV	SWL
<i>RELEASE SAVEPOINT</i>	SQL-transaction	SWV	NS	NS	NS	NS
<i>RETURN</i>	SQL-control	SWV	NS	S	SWL	S
<i>REVOKE</i>	SQL-schema	SWV	SWV	SWV	SWV	SWV
<i>ROLLBACK</i>	SQL-transaction	S	SWL	SWV	SWL	SWV
<i>SAVEPOINT</i>	SQL-transaction	SWV	NS	S	NS	SWL
<i>SELECT</i>	SQL-data	SWV (ANSI joins supported)	SWV (ANSI joins partially supported)	SWV (ANSI joins supported)	SWV (ANSI joins partially supported)	SWV (ANSI joins supported)
<i>SET</i>	SQL-session	S	S	NS	S	S
<i>SET CATALOG</i>	SQL-session	NS	NS	NS	NS	NS
<i>SET COLLATION</i>	SQL-session	NS	NS	NS	NS	NS
<i>SET CONNECTION</i>	SQL-connection	SWL	NS	NS	NS	SWL
<i>SET CONSTRAINT</i>	SQL-connection	NS	NS	SWV	SWV	NS
<i>SET DESCRIPTOR</i>	SQL-session	NS	NS	NS	NS	NS

3.3 SQL Command Reference

ALL/ANY/SOME Operators

The *ALL* operator performs a Boolean test of a subquery for the existence of a value in all rows. The *ANY* operator, and its synonym *SOME*, performs a Boolean test of a subquery for the existence of a value in any of the rows tested.

Platform	Command
DB2	Supported
MySQL	Not supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

SELECT ...

WHERE *expression comparison* {ALL | ANY | SOME} (*subquery*)

Keywords

WHERE *expression*

Tests a scalar expression, such as a column, against every value in the *subquery*, for *ALL*, and against every value until a match is found for *ANY* and *SOME*. All rows must match the expression to return a Boolean TRUE value for the *ALL* operator, while one or more rows must match the expression, for *ANY* and *SOME*, to return a Boolean TRUE value.

comparison

Compares the *expression* to the *subquery*. The *comparison* must be a standard comparison operator like =, <>, !=, >, >=, <, or <=.

Rules at a Glance

The *ALL* operator returns a Boolean TRUE value when one of two things happen: either the subquery returns an empty set (i.e., no records) or every record in the set meets the comparison. *ALL* returns FALSE when any records in the set do not match the value comparison. *ANY* and *SOME* operators return a Boolean TRUE when at least one record in the subquery matches the comparison operation, and FALSE when no record matches the comparison operation (or when a subquery returns an empty result set). If even one return value of the subquery is NULL, then the operation evaluates as NULL, not as TRUE.



Do not include special clauses like *ORDER BY*, *GROUP BY*, *CUBE*, *ROLLUP*, *WITH*, etc. in your subquery. Refer to [SUBQUERY Substatement](#) for more information.

For example, we want to see which authors currently have no titles:

```
SELECT au_id
FROM authors
WHERE au_id <> ALL(SELECT au_id FROM titleauthor)
```

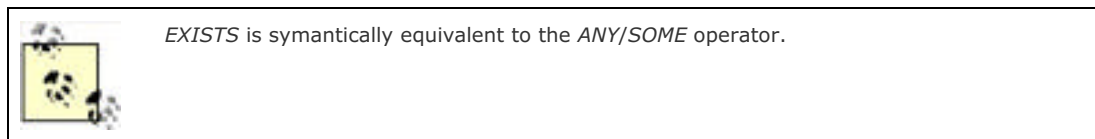
You can use *ANY* / *SOME* to perform filtering checks of different kinds. For example, the following query will retrieve any records from the **employees** table that exist in the **employees_archive** table that were located in Anchorage:

```
SELECT *
FROM employees
WHERE job_lvl = ANY(SELECT job_lvl FROM employees_archive
                    WHERE city = 'Anchorage')
```

The above query returns information about employees who have the same **job_lvl** as any employee working in Anchorage, Alaska.

Programming Tips and Gotchas

The *ALL* and *ANY/SOME* operators are somewhat difficult to get used to. Most developers find it a lot easier to use similar functions like *IN* and *EXISTS*.



Platform Differences

All of the platforms support *ALL* and *ANY/SOME* in the manner described above except MySQL, which doesn't support subqueries until Version 4.0. On pre-Version 4.0 MySQL platforms, consider using the *IN* operator instead of *EXISTS* for *ANY/SOME*.

Oracle supports one minor variation in that you can supply a list of values instead of a subquery. For example, we can find all employees who have a **job_lvl** value equal to 9 or 14:

```
SELECT * FROM employee
WHERE job_lvl = ALL(9, 14);
```

DB2 and SQL Server support some additional comparison operators: not greater than (!>) and not less than (!<).

See Also

- ALL/ANY/SOME
- BETWEEN
- EXISTS
- SELECT
- WHERE

BETWEEN Operator

The *BETWEEN* operator performs a Boolean test of a value against a range of values. It returns TRUE when the value is included in the range and FALSE when the value falls outside of the range. The results are NULL (unknown) if any of the range values are NULL.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

SELECT ...

WHERE *expression* [NOT] BETWEEN *lower_range* AND *upper_range*

Keywords

WHERE *expression*

Compares a scalar expression, such as a column, to the range of values bounded by *upper_range* and *lower_range*.

[NOT] BETWEEN *lower_range* AND *upper_range*

Compares the *expression* to the *lower_range* and *upper_range*. The comparison is inclusive, meaning that it is equivalent to saying "where *expression* is [not] greater than or equal to *lower_range* and less than or equal to *upper_range*."

Rules at a Glance

The *BETWEEN* operator is used to test an expression against a range of values. The *BETWEEN* operator may be used with any datatype except *BLOB*, *CLOB*, *NCLOB*, *REF*, and *ARRAY*.

For example, we want to see **title_ids** that have year-to-date sales between 10,000 and 20,000:

```
SELECT title_id
```

```
FROM titles
```

```
WHERE ytd_sales BETWEEN 10000 AND 20000
```

BETWEEN is inclusive of the range of values listed. In this case, it includes the values of 10,000 and 20,000 in the search. If you want an exclusive search, then you must use the greater than (>) and less than (<) symbols:

```
SELECT title_id
```

```
FROM titles
```

```
WHERE ytd_sales > 10000
```

```
AND ytd_sales < 20000
```

The *NOT* operator allows you to search for values outside of the *BETWEEN* range. For example, you can also find all the **title_ids** that were not published during 2002:

```
SELECT title_id
FROM titles
WHERE pub_date NOT BETWEEN '01-JAN-2003' AND '31-DEC-2003'
```

Programming Tips and Gotchas

Some coders are very particular about how the keyword *AND* is used in *WHERE* clauses. To prevent a casual reviewer from thinking that the *AND* used in a *BETWEEN* operation is a logical *AND* operator, you might want to use parentheses to encapsulate the entire *BETWEEN* clause:

```
SELECT title_id
FROM titles
WHERE (ytd_sales BETWEEN 10000 AND 20000)
AND pubdate >= '1991-06-12 00:00:00.000'
```

Platform Differences

All of the platforms support *BETWEEN* in the manner described above.

See Also

- ALL/ANY/SOME
- EXISTS
- SELECT
- WHERE

CALL Statement

The *CALL* statement invokes a stored procedure.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
CALL procedure_name ([parameter [...] ])
```

Keywords

```
CALL procedure_name
```

Specifies the stored procedure that you want to invoke. It must be a previously defined stored procedure that is available in the current user context (instance, database, schema, etc.).

(*parameter* [,...])

Provides values for each input parameter required by the stored procedure. Each parameter listed is required by the stored procedure in the same ordinal position. Thus, the fifth parameter listed would provide the value for the fifth argument required by the stored procedure. The parameters must be enclosed in parentheses and separated by commas. Note that the parentheses are required even if there are no parameter values (e.g., even if there are no parameter values, you must still use *CALL*()). Strings should be in single quotes. If the stored procedure has only *OUT* parameters, place host variables or parameter markers within the parentheses.

Rules at a Glance

The *CALL* statement makes it easy to invoke a stored procedure. Simply provide the name of the stored procedure and include any parameters used by the stored procedure, enclosing them within parentheses.

This Oracle example creates a simple stored procedure, and then calls it:

```
CREATE PROCEDURE update_employee_salary
```

```
(emp_id NUMBER, updated_salary NUMBER)
```

```
IS
```

```
BEGIN
```

```
    UPDATE employee SET salary = updated_salary
```

```
WHERE employee_id =emp_id ;
```

```
END;
```

```
CALL update_employee_salary(1517, 95000);
```

Programming Tips and Gotchas

The return status of a called stored procedure can be found, typically, by using *GET DIAGNOSTIC*. *GET DIAGNOSTIC* is not widely supported among the various database platforms, so check the platform documentation for more details.

Many platforms also support an alternative command called *EXECUTE* to perform the same functionality. In some cases, you may prefer *EXECUTE* to *CALL*, since the former can be used to execute any kind of prepared SQL, including methods, functions, or batches of SQL code.

DB2

DB2 supports *CALL* along with *DESCRIPTOR* capabilities:

```
CALL procedure_name { [(parameter [,...]) ] | USING DESCRIPTOR  
    descriptor_name }
```

where:

USING DESCRIPTOR *descriptor_name*

Identifies a SQL Descriptor Area (SQLDA) structure containing a valid description of the host variables. An SQLDA is a collection of variables that is required for execution of the SQL *DESCRIBE* statement. The SQLDA variables are options that can be used by the *PREPARE*, *OPEN*, *FETCH*, *EXECUTE*, and *CALL* statements. Values in the SQLDA must be set prior to issuing the *CALL* statement. This is a deprecated feature in DB2. We recommend that you instead use the standard *CALL proc(param, param)* syntax.

DB2 allows you to substitute host variables for the procedure name or the parameters, where appropriate. When using the *USING DESCRIPTOR* clause, the variables must be string variables of 254 or fewer characters.

On DB2, any return status issued by the called stored procedure is returned to the SQLERRD field of the SQLCA (SQL Communications Area). You can also use the *GET DIAGNOSTIC* statement to retrieve the *RETURN_STATUS* value (-1 indicates error) or row count.

MySQL

Not supported.

Oracle

Oracle allows the *CALL* statement to invoke standalone stored procedures, functions, and methods, as well as stored procedures and functions contained within a type or package. Following is the Oracle syntax:

```
CALL [schema][{type_name | package_name}.]procedure_name@dblink  
  
[(parameter [...] )]  
  
[INTO :variable_name [[INDICATOR] :indicator_name ]
```

where:

```
CALL [schema][{type_name | package_name}.] procedure_name@dblink
```

Calls the named object. You may fully enumerate the object name, including schema, type, etc., or allow Oracle to assume the current schema and database instance. If the procedure or function resides in another database, simply specify the database via a database link name, shown as *dblink* in the syntax. *dblink* must refer to a previously created database link.

```
INTO :variable_name
```

Specifies the name of a previously declared variable that will store the value returned when you call a function. *INTO* is required, and only usable, when calling a function.

```
INDICATOR :indicator_name
```

Retains the condition of the host variable, for example whether the return value is NULL, for functions precompiled in a Pro*C/C++ routine.

The parameters used in an Oracle *CALL* statement *may not* include pseudocolumns or the *VALUE* or *REF* functions. You must use a host variable for any parameter that corresponds to an *OUT* or *IN OUT* argument of the called stored procedure.

PostgreSQL

Not Supported.

SQL Server

Not supported. Instead, use the non-ANSI *EXECUTE* statement.

See Also

CREATE/ALTER PROCEDURE

CLOSE CURSOR Statement

The *CLOSE CURSOR* statement closes a server-side cursor created with a *DECLARE CURSOR* statement.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported

PostgreSQL	Supported
SQL Server	Supported, with variations

SQL2003 Syntax

CLOSE cursor_name

Keywords

cursor_name

Names a cursor previously created with the *DECLARE CURSOR* statement.

Rules at a Glance

The *CLOSE* statement closes a cursor and destroys the cursor result set. All the database platforms release any locks that were held by the cursor, though this is not specified in the ANSI standard. For example:

CLOSE author_names_cursor;

Programming Tips and Gotchas

You can also close a cursor implicitly using a *COMMIT* statement or, for cursors defined *WITH HOLD*, using a *ROLLBACK* statement.

DB2

DB2 supports an added optional *WITH* clause on the *CLOSE* statement:

CLOSE cursor_name [WITH RELEASE]

where:

WITH RELEASE

Attempts to release all read locks held by the cursor. Note that DB2 is not always able to release all locks since some locks may be in use by other operations.

The *WITH RELEASE* clause is ignored on cursors run under isolation level 3 CS or UR, when the cursor is defined in a method, when the cursor is defined in a function, or when the cursor is defined in a stored procedure that is called by a function or method. *WITH RELEASE* may allow phantom read and nonrepeatable read anomalies when run under isolation levels RS or RR. For a detailed explanation of isolation levels, including phantom reads and nonrepeatable reads, refer to *SET TRANSACTION ISOLATION LEVEL*.

PostgreSQL

PostgreSQL supports the ANSI standard syntax. PostgreSQL issues an implicit *CLOSE* statement for every open cursor when a transaction is ended with a *COMMIT* or *ROLLBACK* statement.

SQL Server

Microsoft SQL Server supports the ANSI standard syntax, and an additional *GLOBAL* keyword:

CLOSE [GLOBAL] cursor_name

where:

GLOBAL

Identifies the previously defined cursor as a global cursor.

In the ANSI standard behavior, closing a cursor destroys the cursor result set. Locking is a physical feature of each database platform and thus not a part of the ANSI SQL definition. However, all the database platforms covered here drop the locks taken up by the cursor. Another physical implementation detail is that SQL Server does not automatically reallocate memory structures consumed by a cursor to the memory pool. To accomplish such reallocation, you must issue a *DEALLOCATE cursor_name* command.

This example from Microsoft SQL Server opens a cursor and fetches a result set of all employees who have a last name starting with "K":

```
DECLARE employee_cursor CURSOR FOR
```

```
    SELECT lname, fname
```

```
    FROM pubs.dbo.employee
```

```
    WHERE lname LIKE 'K%'
```

```
OPEN employee_cursor
```

```
FETCH NEXT FROM employee_cursor
```

```
WHILE @@FETCH_STATUS = 0
```

```
BEGIN
```

```
    FETCH NEXT FROM employee_cursor
```

```
END
```

```
CLOSE employee_cursor
```

```
DEALLOCATE employee_cursor
```

```
GO
```

See Also

DECLARE CURSOR
FETCH
OPEN

COMMIT Statement

The *COMMIT* statement explicitly ends an open transaction and makes the changes permanent in the database. Transactions can be opened implicitly as part of an *INSERT*, *UPDATE*, or *DELETE* statement, or opened explicitly with a *START* statement. In either case, an explicitly issued *COMMIT* statement will end the open transaction.

Platform	Command
DB2	Supported
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

COMMIT [WORK] [AND [NO] CHAIN]

Keywords

COMMIT [WORK]

Ends the current, open transaction and writes any data manipulated by the transaction to the database. The optional keyword *WORK* is noise and has no effect.

AND [NO] CHAIN

AND CHAIN tells the DBMS to treat the next transaction as if it were a part of the preceding transaction. In effect, the two transactions are separate units of work, but they share a common transaction environment (such as transaction isolation level). Including the optional *NO* keyword tells the DBMS to explicitly use the ANSI default behavior. The *COMMIT* keyword by itself is functionally equivalent to the statement *COMMIT WORK AND NO CHAIN*.

Rules at a Glance

For simple operations, you will execute transactions (that is, SQL code that manipulates or changes data and objects in a database) without explicitly declaring a transaction. However, all transactions are best managed by explicitly closing them with a *COMMIT* statement. Because records and even entire tables can be locked for the duration of a transaction, it is extremely important that transactions are completed as quickly as possible. Thus, manually issuing a *COMMIT* statement with a transaction can help control user concurrency issues and locking problems on the database.

Programming Tips and Gotchas

The most important gotcha to consider is that some database platforms perform automatic and *implicit* transactions, while others require *explicit* transactions. If you assume a platform uses one method of transactions over the other, you may get bitten. Thus, when moving between database platforms you should follow a standard, preset way of addressing transactions. We recommend always using explicit transactions with *START TRAN*, on database platforms that support it, to begin a transaction, and *COMMIT* or *ROLLBACK* to end a transaction.

In addition to finalizing a single or group of data-manipulation operation(s), *COMMIT* has some interesting effects on other aspects of a transaction. First, it closes any associated open cursors. Second, any temporary table(s) specified with *ON COMMIT DELETE ROWS* (an optional clause of the *CREATE TABLE* statement) are cleared of data. Third, all deferred constraints are checked. If any deferred constraints are violated, the transaction is rolled back. Finally, all locks opened by the transaction are released. Please note that SQL2003 dictates that transactions are *implicitly opened* when one of the following statements is executed:

- ALTER
- CLOSE
- COMMIT AND CHAIN
- CREATE
- DELETE
- DROP
- FETCH
- FREE LOCATOR
- GRANT

- *HOLD LOCATOR*
- *INSERT*
- *OPEN*
- *RETURN*
- *REVOKE*
- *ROLLBACK AND CHAIN*
- *SELECT*
- *START TRANSACTION*
- *UPDATE*

If you did not explicitly open a transaction when you started one of the commands listed above, the standard dictates that the DBMS platform open a transaction for you.

DB2

DB2 supports the standard, but not the *AND [NO] CHAIN* clause.

MySQL

MySQL supports *COMMIT* and transactions only with InnoDB tables. It does not support the *AND [NO] CHAIN* clause.

Oracle

Oracle supports the standard, but not the *AND [NO] CHAIN* clause. In addition, Oracle has a couple of extensions to the standard clause:

```
COMMIT [WORK] [ {COMMENT 'text' | FORCE 'text' [, int]} ];
```

where:

COMMENT '*text*'

Associates a comment with the current transaction where '*text*' is a literal string up to 255 characters long. The text string is stored in the Oracle data dictionary view **DBA_2PC_PENDING** with the transaction ID, in case the transaction rolls back.

FORCE '*text*' [, *int*]

Allows an in-doubt distributed transaction to be manually committed, where '*text*' is a literal string that identifies the local or global transaction ID. Transactions can be identified by querying the Oracle data dictionary view **DBA_2PC_PENDING**. The optional *int* parameter is an integer that explicitly assigns a system change number (SCN) to the transaction. Without *int*, the transaction commits using the current SCN.

Issuing a *COMMIT* statement with the *FORCE* clause will commit only the transaction explicitly identified in the *FORCE* clause. It will not affect the current transaction unless it is explicitly identified. The *FORCE* clause is not usable in PL/SQL statements.

The following example commits a transaction while associating a comment with the transaction:

```
COMMIT WORK COMMENT 'In-doubt transaction, Call (949) 555-1234';
```

PostgreSQL

PostgreSQL implements the following syntax:

```
COMMIT [WORK | TRANSACTION];
```

In PostgreSQL, both the *WORK* and *TRANSACTION* keywords are optional. When you issue a *COMMIT*, all open transactions are written to disk and the results of those transactions then become visible to other users. For example:

```
INSERT INTO sales VALUES('7896','JR3435','Oct 28 1997',25,  
    'Net 60','BU7832');  
  
COMMIT WORK;
```

SQL Server

SQL Server does support the *AND [NO] CHAIN* clause. SQL Server supports the keyword *TRANSACTION* as an equivalent to *WORK*, as in the following syntax:

```
COMMIT { [TRAN[SACTION] [transaction_name] ] | [WORK] }
```

Microsoft SQL Server allows the creation of a specific, named transaction using the *START TRAN* statement. The *COMMIT TRANSACTION* syntax allows you to specify an explicit, named transaction to close or to store a transaction name in a variable. Curiously, SQL Server still commits only the most recently opened transaction, despite the name of the transaction that is specified.

When you issue *COMMIT WORK*, SQL Server terminates all open transactions and writes their changes to the database. You *may not* specify a transaction name when using *COMMIT WORK*.

See Also

ROLLBACK
START TRANSACTION

CONNECT

The *CONNECT* statement establishes a connection to the DBMS and to a specific database within the DBMS.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported, with variations

SQL2003 Syntax

```
CONNECT TO {DEFAULT | {[server_name] [AS connection_name]}  
    [USER user_name] }
```

Keywords

DEFAULT

Initiates a default session with the server where the user authorization is the default and the current database is the default. The standard specifies that *CONNECT TO DEFAULT* is issued implicitly if you start a SQL session without first issuing a *CONNECT* statement.

server_name

Establishes a connection to a named server. The *server_name* may be a string literal enclosed in single-quotes or a host variable.

AS *connection_name*

Establishes a connection with the name *connection_name* shown in single-quotes. The optional *connection_name* may be a string literal enclosed in single quotes or a host variable. It is optional only for the first connection attached to a server. All subsequent sessions must include the *AS* clause. This is useful for telling user connections apart when many users, perhaps even a single user, have many open sessions on a given *server_name*.

USER *user_name*

Establishes a connection to the named server under the *user_name* specified.

Rules at a Glance

Issue the *CONNECT* statement to open an interactive SQL session with a DBMS. The period between issuing the *CONNECT* and *DISCONNECT* statements is commonly called a *session*. Typically, you will complete all work on a DBMS during an explicitly invoked session.

If you do not specify the server name, the connection name, or the username, then the platform DBMS will provide default values for you. The defaults vary from platform to platform.

To connect to the "houston" server under a specific user ID, you might issue the command:

```
CONNECT TO houston USER pubs_admin
```

If the DBMS requires named connections, you might use the following, alternative syntax:

```
CONNECT TO houston USER pubs_admin AS  
pubs_administrative_session;
```

If you want just a simple, short-term connection then you might use the default:

```
CONNECT TO DEFAULT;
```

Programming Tips and Gotchas

If the *CONNECT* statement is invoked without explicitly disconnecting, the old session becomes dormant and the new session becomes active. You can then switch between connections using the *SET CONNECTION* statement.



The Oracle tool SQL*Plus uses the *CONNECT* command somewhat differently: to connect a user to a specific schema.

DB2 supports a couple of variations to the ANSI standard:

```
CONNECT { [TO] [{server_name] [IN SHARE MODE |  
    IN EXCLUSIVE MODE [ON SINGLE DBPARTITIONNUM] | [RESET] ] }  
[USER user_name ] { [USING password] | [NEW password  
    CONFIRM password]} }
```

where:

CONNECT

Equates to the SQL statement *CONNECT TO DEFAULT*, when used alone.

TO *server_name*

Identifies the server to which the connection will be established. The *server_name* may be a string literal or a character datatype variable of 8 or fewer characters (UDB) or 16 or fewer characters (OS/390 or z/OS).

IN SHARE MODE

Establishes a concurrent connection and allows other users to connect to the database. Prevents other users from connecting in exclusive mode.

IN EXCLUSIVE MODE [ON SINGLE DBPARTITIONNUM]

Acquires an exclusive connection on the application server and prevents any new users from connecting unless they use the same *user_name* and *password*. The optional *ON SINGLE DBPARTITIONNUM* clause is only usable on a partitioned database. This option specifies that the coordinator database partition is connected in exclusive mode, while all other database partitions are connected in share mode.

RESET

Executes a *COMMIT* statement, then disconnects the session from the server.

USER *user_name*

Establishes the connection under the named user account. The *user_name* may be a string literal or a character datatype variable of 8 or fewer characters in length.

USING *password*

Specifies the *password* for the user account of *user_name*. It may be a string literal or a character datatype variable of 8 or fewer characters in length.

NEW *password* CONFIRM *password*

Specifies a new *password* that will be assigned to the user account called *user_name*. It may be a string literal or a character datatype variable of 18 or fewer characters in length.

For example, we might want to change the password for the user account *sam* to *mephrun* using a *VARCHAR(8)* host variable, *app_server*, to store the value of the application server:

```
CONNECT TO :app_server USER sam NEW 'mephrun' CONFIRM 'mephrun';
```

MySQL

Not supported.

Oracle

The *CONNECT* clause allows a database connection as a specific username. Alternately, a connection can be established for special privileges with *AS SYSOPER* or *AS SYSDBA*. The Oracle syntax is:

```
CONN[ECT] [[username/password] {AS {SYSOPER | SYSDBA} ] ]
```

where:

```
CONN[ECT] username/password
```

Establishes a connection to the database instance.

```
AS { SYSOPER | SYSDBA }
```

Establishes the connection as one of the two optional system roles.

If another connection is already open, *CONNECT* commits any open transactions, closes the current session, and opens the new one.

Oracle also allows the *CONNECT* statement in its SQL*Plus and iSQL*Plus tools.

PostgreSQL

PostgreSQL does not explicitly support the *CONNECT* statement. However, it does support the statement *SPI_CONNECT* under the Server Programming Interface and *PG_CONNECT* under the PG/TCL programming package.

SQL Server

SQL Server supports the basic elements of the *CONNECT* statement within Embedded SQL (inside of C++ or Visual Basic programs), as in the following:

```
CONNECT TO [server_name.]database_name [AS connection_name]
```

```
USER {login_name[.password] | $integrated}
```

where:

```
CONNECT TO server_name.database_name
```

Specifies the server and database to which you want a connection. You may leave off the *server_name* if you wish to default to the local server.

```
AS connection_name
```

Names the connection as an alphanumeric string up to 30 characters in length. Symbols are allowed, except for hyphens (-), but the first character must be a letter. *CURRENT* and *ALL* are reserved and may not be used for a *connection_name*. You need only define connection names when making more than one connection.

```
USER {login_name[.password] | $integrated}}
```

Connects the session under the *login_name* specified using the *password* provided or using Windows integrated security. The password is optional when supplying a login name.

For example, we can connect to the server named *new_york* as the Windows login *pubs_admin*:

```
CONNECT TO new_york.pubs USER pubs_admin
```


To issue the same command under SQL Server standard security:

```
EXEC SQL CONNECT TO new_york.pubs USER pubs_admin
```

To issue the same command under Windows integrated security:

```
EXEC SQL CONNECT TO new_york.pubs USER $integrated
```

To switch to a different connection, you should use the *SET CONNECTION* statement.

See Also

SET CONNECTION

CREATE/ALTER DATABASE Statement

The ANSI standard does not actually contain a *CREATE DATABASE* statement. However, since it is nearly impossible to operate a SQL database without this command, we've added *CREATE DATABASE* here. Almost all database platforms support some version of this command.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

Rules at a Glance

This command creates a new, blank database with a specific name. Most DBMS platforms require the user to possess administrator privileges in order to create a new database. Once the new database is created, you can populate it with database objects (such as tables, views, triggers, and so on) and populate the tables with data.

Depending on the platform, *CREATE DATABASE* may also create corresponding files on the filesystem that contain the data and metadata of the database.

Programming Tips and Gotchas

Since *CREATE DATABASE* is not an ANSI command, it is prone to rather extreme variation in syntax between platforms.

DB2

The DB2 statement *CREATE DATABASE* initializes a new database with a variety of user-defined characteristics, such as collation and autoconfiguration options. In addition, the statement also creates the three initial tablespaces, system tables, and recovery logs required by a DB2 database:

```
CREATE {DATABASE | DB} database_name
```

```
[AT DBPARTITIONNUM] [ON path_and_drive] [ALIAS database_alias]
```

```
[USING CODESET codeset TERRITORY territory]
```

```
[COLLATE USING {COMPATIBILTiy | IDENTITY | NLSCHAR | SYSTEM} ]
```

```
[NUMSEGS int] [DFT_EXTENT_SZ int]
```

```
[CATALOG TABLESPACE tablespace_definition]
```

[USER TABLESPACE *tablespace_definition*]

[TEMPORARY TABLESPACE *tablespace_definition*] [WITH "comments"]

[AUTOCONFIGURE [USING *param value* [...]]

[APPLY {DB ONLY | DB AND DBM | NONE}]

where:

CREATE {DATABASE | DB} *database_name*

Creates a database called *database_name*. The name must be unique from all other database names on either the local database directory or the system database directory.

AT DBPARTITIONNUM

Creates the database on the database partition where the command was issued. This command is used only to recreate a database that was dropped; it should not be used to create a brand new database. Once the *CREATE DATABASE AT DBPARTITIONNUM* statement is issued, the database must be immediately restored to the node.

ON *path_and_drive*

Specifies the path and drive, up to 205 characters long, to create the database on a Windows or Unix-based system. When this clause is omitted, the database is created on the default database path declared in the database manager configuration file *dftdbpath* parameter.

ALIAS *database_alias*

Declares an alias for the database within the system database directory.

USING CODESET *codeset* TERRITORY *territory*

Declares the *codeset* and *territory* used by the database. Once these values are set, they cannot be changed later.

COLLATE USING {COMPATIBLITY | IDENTITY | NLSCHAR | SYSTEM}

Specifies the type of collation used for the database. Once set, the collation sequence cannot be changed later. Options are:

COMPATIBILITY

Uses DB2 v2 collation sequence for backward compatibility.

IDENTITY

Uses the identity collation sequence, where strings are compared byte for byte.

NLSCHAR

Uses the collation sequence unique for the database codeset and territory. This option is currently only supported with the Thai code page (CP874).

SYSTEM

Uses the collation sequence for the current territory.

NUMSEGS *int*

Specifies the number of segment directories, as *int*, created to store DAT, IDX, LF, LB, and LBA files for any default System Managed Space (SMS) tablespaces. This setting does not affect DMS tablespaces, SMS tablespaces with explicitly defined creation characteristics, or SMS tablespaces created after the database is created.

DFT_EXTENT_SZ *int*

Specifies the default extent size, as *int*, of the tablespaces in the database.

CATALOG TABLESPACE *tablespace_definition*

Specifies the definition of the tablespace that holds the catalog tables. Details on the syntax for *tablespace_definition* are detailed below. When this clause is omitted, the catalog tables will be created in an SMS tablespace called **SYSCATSPACE** with the values set by the *NUMSEGS* and *DFT_EXTENT_SZ* clauses.

USER TABLESPACE *tablespace_definition*

Specifies the definition of the tablespace that holds the initial user tablespace. Details on the syntax for *tablespace_definition* are detailed below. When this clause is omitted, the user tablespace will be created in an SMS tablespace called **USERSPACE1** with the values set by the *NUMSEGS* and *DFT_EXTENT_SZ* clauses.

TEMPORARY TABLESPACE *tablespace_definition*

Specifies the definition of the tablespace that holds the initial system temporary tablespace. Details on the syntax for *tablespace_definition* are detailed below. When this clause is omitted, the user tablespace will be created in an SMS tablespace called **TEMPSPACE1** with the values set by the *NUMSEGS* and *DFT_EXTENT_SZ* clauses.

WITH "*comments*"

Specifies a description, up to 30 characters in length, for the database within the database directory.

AUTOCONFIGURE [USING *param value* [...]]

Declares user-defined settings for important database settings such as the buffer pool size. When issued without the *USING* clause, DB2 calculates the optimal settings and applies them. The list of *param* settings and their possible *values* are:

mem_percent

Defines the percentage of memory dedicated to the database. The default is 25, but may range from 1-100. Do not set this to 100 if other applications are running on the server.

workload_type

Defines the database workload as either '*simple*', '*mixed*', or '*complex*', with '*mixed*' as the default. The '*simple*' workload tunes I/O for transactions, while '*complex*' tunes the workload for more CPU intensive queries. '*Mixed*' provides tuning that is effective, though not optimized, for both workloads.

num_stmts

Defines the number of statements per unit of work. The default is 25, but may range from 1-1,000,000.

tpm

Defines the expect number of transactions per minute. The default is 60, but may range from 1-50,000.

admin_priority

Defines whether the database should be tuned for better performance (*value* = '*performance*') or better

recovery time (*value* = 'recovery'), or the middle ground (*value* = 'both'), which is the default.

num_local_apps

Defines the number of connected local applications. The default is 0, but may range from 0-5,000.

num_remote_apps

Defines the number of connections by remote applications. The default is 100, but may range from 0-5,000.

isolation

Defines the transaction isolation level of the applications connecting to the database. The default is RR, but valid values include RR, RS, CS, and UR. Refer to the entry on *SET TRANSACTION ISOLATION LEVEL* for more details on DB2's isolation levels.

bp_resizeable

Defines whether buffer pools may be resized or not. The default is yes, but it may also be set to no.

APPLY {DB ONLY | DB AND DBM | NONE}

Determines the scope of any autoconfiguration parameters set using the *AUTOCONFIGURATION* clause:

DB ONLY

Applies changes only to the database configuration and buffer pool settings.

DB AND DBM

Applies settings to the database manager configuration, the database configuration, and the buffer pool settings.

NONE

Displays the recommended settings, but does not apply them.

The *tablespace_definition* used by DB2 has its own syntax as well:

MANAGED BY

```
{SYSTEM USING ('container' [...]) |  
DATABASE USING ({FILE | DEVICE} 'container' int [...]) }  
[EXTENTSIZE int] [PREFETCHSIZE int] [OVERHEAD int]  
[TRANSFERRATE int]
```

where:

MANAGED BY

Defines whether the tablespace is managed by a system managed space (SMS) or a database managed space (DMS):

SYSTEM USING ('container' [...])

Defines one or more *containers* for the SMS of up to 240 bytes in length. The *container* may be an absolute or relative directory name.

DATABASE USING ({FILE | DEVICE} '*container*' *int* [,...])

Defines one or more *containers* for the DMS of up to 254 bytes in length. The *container* may be an absolute or relative directory name. The type of container (*FILE* for a filename or *DEVICE* for a pre-existing device) and its size (*int*), plus a suffix of K (kilobytes), M (megabytes), or G (gigabytes) are also specified. If the file does not exist when specified, DB2 will create one. The device must already exist to be used.

EXTENTSIZE *int*

Defines the number of pages, *int*, that are written to the container before skipping to the next container. (The size of the pages is determined by the *PAGESIZE* setting.) Alternately, the extent size (*int*) may be declared in bytes by attaching a suffix of K (kilobytes), M (megabytes), or G (gigabytes) to the integer value.

PREFETCHSIZE *int*

Defines the number of pages, *int*, that are read from the tablespace when prefetching data. (The size of the pages are determined by the *PAGESIZE* setting.) Alternately, the extent size (*int*) may be declared in bytes by attaching a suffix of K (kilobytes), M (megabytes), or G (gigabytes) to the integer value.

OVERHEAD *int*

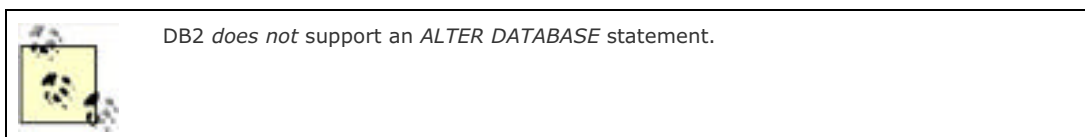
Defines the I/O controller overhead, disk seek, and latency time, in milliseconds, *int*, used to determine the cost of I/O during query optimization. The value for *int* should be an average for all containers that belong to the tablespace.

TRANSFERRATE *int*

Defines the time to read one page into memory, in milliseconds, *int*, used to determine the cost of I/O during query optimization. The value for *int* should be an average for all containers that belong to the tablespace.

Note that DB2 tablespaces may also be created using the DB2-specific command *CREATE TABLESPACE*.

On DB2 servers that are enabled to support Lightweight Directory Access Protocol (LDAP), the new database is automatically registered in the LDAP directory.



The following example creates a database called **sales** with specifically declared *NUMSEGS*, *DFT_EXTENT_SZ*, and *AUTOCONFIGURATION* enabled and applied:

```
CREATE DATABASE sales
NUMSEGS 200 DFT_EXTENT_SZ 1000
WITH "Temporary sales database"
AUTOCONFIGURE APPLY DB ONLY;
```

The *WITH* clause adds a description of the **sales** database to the database directory.

MySQL

In MySQL, *CREATE DATABASE* essentially creates a new directory that holds the database objects:

```
CREATE DATABASE [IF NOT EXISTS] database_name
```

where:

CREATE DATABASE *database_name*

Creates a database and directory of *database_name*. The database directory appears under the MySQL data directory. Tables in MySQL then appear as files in the database directory.

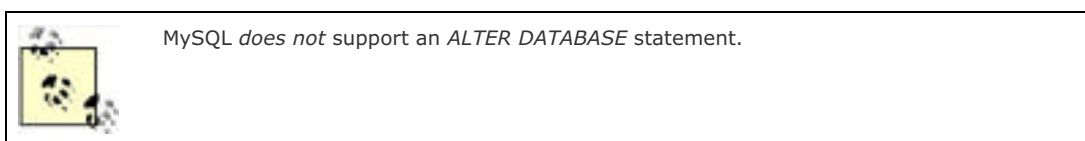
IF NOT EXISTS

Avoids an error if the database already exists.

For example, we could create a database called **sales_archive** on a MySQL server. If the database already exists, we want to abort the command without causing an error:

CREATE DATABASE IF NOT EXISTS sales_archive

Any tables that we create in the **sales_archive** database will now appear as files within the **sales_archive** directory.



Oracle

Oracle provides an extraordinary level of control over database file structures beyond merely naming the database and specifying a path for the database files. *CREATE* and *ALTER DATABASE* are very powerful commands in Oracle. Some of the more sophisticated clauses are best used only by experienced DBAs. These commands can be very large and complex. (*ALTER DATABASE* alone consumes over 50 pages in the Oracle vendor documentation!)

Novices should be aware that *CREATE DATABASE*, when run, erases all data in the specified datafiles that are already in existence. Similarly, all the data in an existing database will be lost.

Following is the syntax to create a new database in Oracle:

```
CREATE DATABASE [database_name]  
{[USER SYS IDENTIFIED BY password | USER SYSTEM IDENTIFIED BY password]}  
[CONTROLFILE REUSE]  
{[LOGFILE definition [,... ] [MAXLOGFILES int] [[MAXLOGMEMBERS] int]  
  [[MAXLOGHISTORY] int] [{ARCHIVELOG | NOARCHIVELOG}] [FORCE LOGGING]}  
[MAXDATAFILES int]  
[MAXINSTANCES int]  
[CHARACTER SET charset]  
[NATIONAL CHARACTER SET charset]  
[EXTENT MANAGEMENT {DICTIONARY | LOCAL}]  
[DATAFILE definition [,... ]]  
[SYSAUX DATAFILE definition [,... ]]  
[DEFAULT TABLESPACE tablespace_name  
  [DATAFILE file_definition]  
  EXTENT MANAGEMENT {DICTIONARY |  
    LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M] } } ]
```

```
[ [{ BIGFILE | SMALLFILE}] DEFAULT [TEMPORARY] TABLESPACE tablespace_name
    [TEMPFILE file_definition]
    EXTENT MANAGEMENT {DICTIONARY |
        LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M] ] } } ]
[ [{ BIGFILE | SMALLFILE}] UNDO TABLESPACE tablespace_name
    [DATAFILE temp_datafile_definition] ]
[SET TIME_ZONE = ' { {+ | -} hh:mi | time_zone_region}' ]
[SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE]
```

Following is the syntax to alter an existing database:

```
ALTER DATABASE [database_name]
[ARCHIVELOG | NOARCHIVELOG] |
{MOUNT [{STANDBY | CLONE} DATABASE] |
    OPEN [READ WRITE | READ ONLY
        [RESETLOGS | NORESETLOGS] | [UPGRADE | DOWNGRADE] ] } |
{ACTIVATE [PHYSICAL | LOGICAL] STANDBY DATABASE
    [SKIP [STANDBY LOGFILE] ] |
    SET STANDBY [DATABASE] TO MAXIMIZE {PROTECTION | AVAILABILITY |
        PERFORMANCE} |
    REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE ['file']
        [FOR logminer_session_name] |
    {COMMIT | PREPARE} TO SWITCHOVER TO
        {[PHYSICAL | LOGICAL] PRIMARY | STANDBY} [WITH[OUT]
            SESSION SHUTDOWN] [WAIT | NOWAIT]} |
    CANCEL} |
    START LOGICAL STANDBY APPLY [IMMEDIATE] [NODELAY]
        [{INITIAL int | NEW PRIMARY dblink_name | {FINISH |
            SKIP FAILED TRANSACTION] } ] |
    {STOP | ABORT} LOGICAL STANDBY APPLY |
    [[NO]PARALLEL int] } |
{RENAME GLOBAL_NAME TO database [.domain [.domain ...]] |
    CHARACTER SET character_set |
    NATIONAL CHARACTER SET character_set |
    DEFAULT TABLESPACE tablespace_name |
    DEFAULT TEMPORARY TABLESPACE {GROUP int | tablespace_name} |
    {DISABLE BLOCK CHANGE TRACKING | ENABLE BLOCK CHANGE TRACKING [USING
        FILE 'file' [REUSE] ] } |
    FLASHBACK {ON | OFF} |
    SET TIME_ZONE = ' { {+ | -} hh:mi | time_zone_region}' |
```

```
SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE } |
{ENABLE | DISABLE} { [PUBLIC] THREAD int | INSTANCE 'instance_name' } |
{GUARD {ALL | STANDBY | NONE} } |
{CREATE DATAFILE 'file' [,...] [AS {NEW | file_definition [,...]} ] |
  {DATAFILE 'file' | TEMPFILE 'file' [,...]}
  {ONLINE | OFFLINE [FOR DROP | RESIZE int [K | M] |
    END BACKUP | AUTOEXTEND {OFF | ON [NEXT int [K | M] [MAXSIZE
    [UNLIMITED | int [K | M] ] } |
  {TEMPFILE 'file' | TEMPFILE 'file' [,...]}
  {ONLINE | OFFLINE | DROP [INCLUDING DATAFILES] |
    RESIZE int [K | M] | AUTOEXTEND {OFF | ON [NEXT int [K | M]
    [MAXSIZE [UNLIMITED | int [K | M] ] } |
  RENAME FILE 'file' [,...] TO 'new_file_name' [,...] } |
{[[NO] FORCE LOGGING] | [ [NO]ARCHIVELOG [MANUAL] ] |
  [ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE |
  FOREIGN KEY) [,...]] COLUMNS |
[ADD | DROP] [STANDBY] LOGFILE
  {[THREAD int | INSTANCE 'instance_name']} {[GROUP int |
  logfile_name [,...]} [SIZE [K | M]] | [REUSE] |
  [MEMBER] 'file' [REUSE] [,...] TO logfile_name [,...]} |
ADD LOGFILE MEMBER 'file' [REUSE] [,...] TO {[GROUP int |
  logfile_name [,...]} |
DROP [STANDBY] LOGFILE {MEMBER 'file' | {[GROUP int | logfile_name
  [,...]} }
CLEAR [UNARCHIVED] LOGFILE {[GROUP int | logfile_name [,...]} [,...]}
  [UNRECOVERABLE DATAFILE] } |
{CREATE [LOGICAL | PHYSICAL] STANDBY CONTROLFILE AS 'file' [REUSE] |
  BACKUP CONTROLFILE TO
  {'file' [REUSE] | TRACE [AS 'file' [REUSE] ] {RESETLOGS |
  NORESETLOGS}} } |
{RECOVER
  {[AUTOMATIC [FROM 'location']] |
  {[STANDBY] DATABASE
  {[UNTIL {CANCEL | TIME date | CHANGE int} |
  USING BACKUP CONTROLFILE } |
  {[STANDBY] {TABLESPACE tablespace_name [,...]} | DATAFILE
  'file' [,...]} [UNTIL [CONSISTENT WITH] CONTROLFILE]} |
  TABLESPACE tablespace_name [,...]} | DATAFILE 'file' [,...]} |
  LOGFILE filename [,...]} {[TEST | ALLOW int CORRUPTION | [NO]PARALLEL
```



```
int}} |  
CONTINUE [DEFAULT] |  
CANCEL [IMMEDIATE][NOWAIT] }  
{MANAGED STANDBY DATABASE  
{[[NO]TIMEOUT int] [{[NO]DELAY int | DEFAULT DELAY}] [NEXT int]  
[[NO]EXPIRE int] [{[NO]PARALLEL int}] [USING CURRENT LOGFILE]  
[UNTIL CHANGE int]  
[THROUGH  
{ALL ARCHIVELOG | {ALL | LAST | NEXT} SWITCHOVER |  
[[THREAD int] SEQUENCE int] } |  
[CANCEL [IMMEDIATE] [{WAIT | NOWAIT}]] |  
[DISCONNECT [FROM SESSION] [{[NO]PARALLEL int}]  
FINISH [SKIP [STANDBY LOGFILE]] [{WAIT | NOWAIT}]] |  
  
{BEGIN | END} BACKUP} }
```

The syntax elements in Oracle are as follows:

{CREATE | ALTER} DATABASE *database_name*

Creates or alters a database with the name *database_name*. The database name can be up to 8 bytes in length and may not contain European or Asian characters. You can omit a database name and allow Oracle to create a database name for you, but beware that Oracle can create some counter-intuitive names.

USER SYS IDENTIFIED BY *password*

USER SYSTEM IDENTIFIED BY *password*

Specifies passwords for the *SYS* and *SYSTEM* users. You may specify neither or both of these clauses, but not just one of them.

CONTROLFILE REUSE

Causes existing control files to be reused, enabling you to specify existing files in the *CONTROL_FILES* parameter in *INIT.ORA*. Oracle will then overwrite any information those files may contain. This clause is normally used when recreating a database. Consequently, you probably don't want to use this clause in conjunction with *MAXLOGFILES*, *MAXLOGMEMBER*, *MAXLOGHISTORY*, *MAXDATAFILES*, or *MAXINSTANCES*.

LOGFILE *definition*

Specifies one or more logfiles for the database. You may define multiple files all with the same size and characteristics, using the *file* parameter, or you may define multiple files, each with its own size and characteristics. The entire logfile definition syntax is rather ponderous, but offers a great deal of control:

```
LOGFILE { ('file' [...]) [SIZE int [K | M]
```

```
[GROUP int] [REUSE]} [...]
```

where:

LOGFILE ('*file*' [,...])

Defines one or more files that will act as redo logfiles; *file* is both the filename and path. Any files defined in the *CREATE DATABASE* statement are assigned to redo log thread number 1. When specifying multiple redo logfiles, each filename should be enclosed in single quotes and separated from the other names by commas. The entire list should be enclosed in parentheses.

SIZE *int* [K | M]

Specifies the size of the redo logfile in bytes as an integer value, *int*. Alternately, you may define the redo logfile in large units than bytes by appending a K (for kilobytes) or an M (for megabytes).

GROUP *int*

Defines the integer ID, *int*, of the redo logfile group. The value may be from 1 to the value of the *MAXLOGFILES* clause. An Oracle database must have at least two redo logfile groups. Oracle will create a redo logfile group for you, at 100 MB each, if you leave off this group ID.

REUSE

Reuses an existing redo logfile.

MAXLOGFILES *int*

Sets the maximum number of logfiles, *int*, available to the database being created. The minimum, maximum, and default for this clause are OS-dependent.

MAXLOGMEMBERS *int*

Sets the maximum number of members (i.e., copies) for a redo logfile group. The minimum value is 1 while the maximum and default values for this clause are OS-dependent.

MAXLOGHISTORY *int*

Sets the maximum number of archived redo logfiles available to a Real Application Cluster. You can use the *MAXLOGHISTORY* clause only when Oracle is in *ARCHIVELOG* mode on a Real Application Cluster. The minimum value is 0 while the maximum and default values for this clause are OS-dependent.

ARCHIVELOG | NOARCHIVELOG

Defines how redo logs operate. When used with *ALTER DATABASE*, allows the current setting to be changed. *ARCHIVELOG* saves data stored in the redo log(s) to an archiving file, providing for media recoverability. Conversely, *NOARCHIVELOG* allows a redo log to be reused without archiving the contents. Both options provide recoverability, although *NOARCHIVELOG* (the default) does not provide media recovery.

FORCE LOGGING

Places all instances of the database into *FORCE LOGGING* mode, in which all changes to the database are logged, except for changes to temporary tablespaces and segments. This setting takes precedence over any tablespace- or object-level settings.

MAXDATAFILES *int*

Sets the initial number of datafiles, *int*, available to the database being created. Note that the *INIT.ORA* setting, **DB_FILES**, also limits the number of datafiles accessible to the database instance.

MAXINSTANCES *int*

Sets the maximum number of instances, *int*, which may mount and open the database being created. The minimum value is 1 while the maximum and default values for this clause are OS-dependent.

CHARACTER SET *charset*

Controls the language character set in which the data is stored. The value for *charset* cannot be AL16UTF16. The value of *charset* is OS-dependent.

NATIONAL CHARACTER SET *charset*

Controls the national language character set for data stored in *NCHAR*, *NCLOB*, and *NVARCHAR2* columns. The value for *charset* must be either AL16UTF16 (the default) or UTF8.

EXTENT MANAGEMENT {*DICTIONARY* | *LOCAL*}

Creates a locally managed *SYSTEM* tablespace, otherwise the *SYSTEM* tablespace will be dictionary managed. This clause requires a default temporary tablespace. If you omit a *DATAFILE* clause, you can also omit a default temporary tablespace because Oracle will create them both for you.

DATAFILE *definition*

Specifies one or more datafiles for the database. (All these datafiles become part of the *SYSTEM* tablespace.) You may repeat filenames to define multiple files all with the same size and characteristics. Alternately, you may repeat the entire *DATAFILE* clause, with each occurrence defining one or more files with the same size and characteristics. The entire datafile definition syntax is rather large, but offers a great deal of control:

```
DATAFILE {'file1' [,...]} [GROUP int] [SIZE int [K | M] [REUSE]
[AUTOEXTEND [OFF | ON [NEXT int [K | M]
[MAXSIZE [UNLIMITED | int [K | M]]] } [,...]
```

where:

```
DATAFILE ('file1' [,...])
```

Defines one or more files that will act as the datafile(s), where *file1* is both the filename and path. For multiple files, each filename should be enclosed in single-quotes and separated by a comma. The entire list should be enclosed in parentheses.

```
GROUP int
```

Defines the integer ID, *int*, of the datafile group. The value may be from 1 to the value of *MAXLOGFILES* clause. An Oracle database must have at least two datafile groups. Oracle will create them for you, at 100 MB each, if you leave off this clause.

```
SIZE int [K | M]
```

Specifies the size of the datafile in bytes as an integer value, *int*. Alternately, you may define the datafile in large units by appending a K (for kilobytes) or an M (for megabytes).

```
REUSE
```

Reuses the existing datafile.

```
AUTOEXTEND
```

Enables (*ON*) the automatic extension of new or existing data or temp files (but does not redo logfiles). *NEXT* specifies the next increment of space allocated to the file in bytes, kilobytes (*K*), or megabytes (*M*) when more space is needed.

```
MAXSIZE [UNLIMITED | int [K | M]]
```

Specifies the maximum disk space allowed for automatic extension of the file. *UNLIMITED* allows the file to grow without an upper limit, except of course the total capacity of the drive. Otherwise, you may define the maximum size limit as an integer, *int*, in bytes (the default), kilobytes with the keyword *K*, and megabytes with the keyword *M*.

SYSAUX DATAFILE *definition*

Specifies one or more datafiles for the *SYSAUX* tablespace. By default, Oracle creates and manages the *SYSTEM* and *SYSAUX* tablespaces automatically. You must use this clause if you have specified a datafile for the *SYSTEM* tablespace. If you omit the *SYSAUX* clause when using Oracle-managed files, then Oracle will create the *SYSAUX* tablespace as an online, permanent, locally managed tablespace with a single datafile of 100 MB, using automatic segment-space management and logging.

BIGFILE | *SMALLFILE*

Specifies the default file type of a subsequently created tablespace. *BIGFILE* indicating the tablespace will contain a single datafile or tempfile of up to 8 exabytes (8 million terabytes) in size, while *SMALLFILE* indicates the tablespace is a traditional Oracle tablespace. The default, when omitted, is *SMALLFILE*.

DEFAULT TABLESPACE *tablespace_name*

Specifies a default permanent tablespace for the database for all non-*SYSTEM* users. When this clause is omitted, the *SYSTEM* tablespace is the default permanent tablespace for non-*SYSTEM* users. The *EXTENT MANAGEMENT* clause has the same syntax as in the *DEFAULT TEMPORARY TABLESPACE* clause appearing later.

DEFAULT TEMPORARY TABLESPACE *tablespace_name* [TEMPFILE *file_definition*]

Defines the name and location of the default temporary tablespace for the database. Users who are not explicitly assigned to a temporary tablespace will operate in this one. If you don't create a default temporary tablespace, Oracle uses the *SYSTEM* tablespace. Under *ALTER DATABASE*, this clause allows you to change the default temporary tablespace.

TEMPFILE *file_definition*

The tempfile definition is optional when the *DB_CREATE_FILE_DEST INIT.ORA* parameter is set. Otherwise, you'll have to define the tempfile yourself. The *TEMPFILE* definition syntax is otherwise identical to the *DATAFILE* definition syntax earlier in this section.

EXTENT MANAGEMENT {DICTIONARY | LOCAL {AUTOALLOCATE | UNIFORM [SIZE *int* [K | M]}}

Defines the way in which the *SYSTEM* tablespace is managed. When this clause is omitted, the *SYSTEM* tablespace is dictionary-managed. Once created as a locally managed tablespace, it cannot be converted back to a dictionary-managed tablespace, nor can any new dictionary-managed tablespaces be created in the database.

DICTIONARY

Specifies that the Oracle data dictionary manages the tablespace. This is the default. The *AUTOALLOCATE* and *UNIFORM* subclauses are not used with this clause.

LOCAL

Declares that the tablespace is locally managed. This clause is optional since all temporary tablespaces have locally managed extents by default. Use of this clause requires a default temporary tablespace. If you do not manually create one, Oracle will automatically create one called *TEMP* of 10 MB size with autoextend disabled.

AUTOALLOCATE

Specifies that new extents will be allocated as needed by the locally managed tablespace.

UNIFORM [SIZE *int* [K | M]]

Specifies that all extents of the tablespace are the same size (*UNIFORM*), in bytes, as *int*. The *SIZE* clause allows you to configure the size of the extents to your liking in bytes (the default), in kilobytes using the keyword *K*, or in megabytes using the keyword *M*. The default is 1M.

UNDO TABLESPACE *tablespace_name* [DATAFILE *temp_datafile_definition*]

Defines the name and location for undo data, creating a tablespace named *tablespace_name*, but only if you have set the *UNDO_MANAGEMENT INIT.ORA* parameter to *AUTO*. If you don't use this clause, Oracle manages undo space via rollback segments. (You may also set the *INIT.ORA* parameter to *UNDO_TABLESPACE*. If so, the value of the parameter and the *tablespace_name* used here must be identical.)

where:

DATAFILE *temp_datafile_definition*

Creates and assigns the datafile, as you have defined it, to the undo tablespace. Refer to the earlier section on *DATAFILE* for the full syntax on this clause. This clause is required if you have not specified a value for the *INIT.ORA* parameter *DB_CREATE_FILE_DEST*.

SET TIME_ZONE = ' { {+ | -} *hh:mi* | *time_zone_region* }'

Sets the time zone for the database, either by specifying a delta from Greenwich Mean Time (now called Coordinated Universal Time) or by specifying a time zone region. (For a list of time zone regions, query the **tzname** column of the *v\$timezone_names* view.) If you do not use this clause, Oracle defaults to the operating-system time zone.

SET DEFAULT TABLESPACE

Sets all tablespaces created by the current *CREATE DATABASE* or *ALTER DATABASE* statement as either *BIGFILE* or *SMALLFILE*. When creating databases, this clause also applies to the *SYSTEM* and *SYSAUX* tablespaces.

MOUNT [{*STANDBY*} | *CLONE*] DATABASE]

Mounts a database for users to access. The *STANDBY* keyword mounts a physical standby database, enabling it to receive archived redo logs from the primary instance. The *CLONE* keyword mounts a clone database. This clause cannot be used with *OPEN*.

OPEN [*READ WRITE* | *READ ONLY*] [*RESETLOGS* | *NORESETLOGS*] [*UPGRADE* | *DOWNGRADE*]

Opens the database separately from the mounting process. (Mount the database first.) *READ WRITE* opens the database in read/write mode, allowing users to generate redo logs. *READ ONLY* allows reads but disallows changes to redo logs. *RESETLOGS* discards all redo information not applied during recovery and sets the log sequence number to 1. *NORESETLOGS* retains the logs in their present condition. The optional *UPGRADE* and *DOWNGRADE* clauses tell Oracle to dynamically modify the system parameters as required for database upgrade or downgrade, respectively. The default is *OPEN READWRITE NORESETLOGS*.

ACTIVATE [*PHYSICAL* | *LOGICAL*] STANDBY DATABASE [*SKIP* [*STANDBY LOGFILE*]]

Promotes a standby database to the primary database. You can optionally specify a *PHYSICAL* standby, the default, or a *LOGICAL* standby. Use the *SKIP* clause to immediately promote a physical standby and discard any data still unapplied by the *RECOVER MANAGED STANDBY DATABASE FINISH* statement. The clause *STANDBY LOGFILE* is noise.

SET STANDBY [*DATABASE*] TO MAXIMIZE {*PROTECTION* | *AVAILABILITY* | *PERFORMANCE*}

Sets the level of protection for data in the primary database. The old terms *PROTECTED* and *UNPROTECTED* equate to *TO MAXIMIZE PROTECTION* and *MAXIMIZE PERFORMANCE*, respectively, where:

PROTECTION

Provides the highest level of data protection, but has the greater overhead and negatively impacts availability. This setting commits transactions only after all data necessary for recovery has been physically written in at least one physical standby database that uses *SYNC* log transport mode.

AVAILABILITY

Provides the second highest level of data protection, but the highest level of availability. This setting commits transactions only after all data necessary for recovery has been physically written in at least one physical or logical standby database that uses SYNC log transport mode.

PERFORMANCE

Provides the highest level of performance, but compromises data protection and availability. This setting commits transactions before all data necessary for recovery has been physically written to a standby database.

REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE [*file*]

Manually registers redo logfiles from a failed primary server when issued from a standby server. The logfile may optionally be declared as *PHYSICAL* or *LOGICAL*. The *OR REPLACE* clause allows updates to details for an existing archivelog entry.

FOR *logminer_session_name*

Registers the logfile with a single, specific LogMiner session in an Oracle Streams environment.

{*COMMIT* | *PREPARE*} TO SWITCHOVER TO {[*PHYSICAL* | *LOGICAL*] *PRIMARY* | *STANDBY*}

Performs a graceful switchover, moving the current primary database to standby status and promoting a standby database to primary. (In a RAC environment, all instances other than the current instance have to be shutdown.) To gracefully switch over, you should issue the command twice, once to prepare the primary and standby databases to begin exchanging logfiles in advance of the switchover using *PREPARE TO SWITCHOVER*. To demote the primary database and switchover of the standby databases, use *COMMIT TO SWITCHOVER*. The *PHYSICAL* clause puts the primary database into physical standby mode. The *LOGICAL* clause puts the primary database into logical standby mode. However, you must then issue an *ALTER DATABASE START LOGICAL STANDBY APPLY* statement.

WITH[OUT] *SESSION SHUTDOWN*] [*WAIT* | *NOWAIT*]

WITH SESSION SHUTDOWN closes any open application sessions and rolls back any uncommitted transactions during a switchover of physical databases (but not logical ones). *WITHOUT SESSION SHUTDOWN*, the default, causes the *COMMIT TO SWITCHOVER* statement to fail if it encounters any open application sessions. *WAIT* returns control to the console after completion of the *SWITCHOVER* command, while *NOWAIT* returns control before the command completes.

START LOGICAL STANDBY APPLY [*IMMEDIATE*][*NODELAY*][{*INITIAL int* | *NEW PRIMARY dblink_name*} | {*FINISH* | *SKIP*} *FAILED TRANSACTIONS*}]

Starts to apply redo logs to the logical standby database. *IMMEDIATE* tells the Oracle LogMiner to read the redo data in the standby redo logfiles. *NODELAY* tells Oracle to ignore a delay for the apply, such as when the primary database is unavailable or disabled. *INITIAL* is used the first time you apply logs to the standby database. *NEW PRIMARY* is required after a switchover has completed or after a standby database has processed all redo logs and another standby is promoted to primary. Use *SKIP FAILED TRANSACTION* to skip the last transaction and to restart the apply. Use *FINISH* to apply the data in the redo logs if the primary database is disabled.

[*STOP* | *ABORT*] LOGICAL STANDBY APPLY

Stops the application of redo logs to a logical standby server. *STOP* performs an orderly stop, while *ABORT* performs an immediate stop.

[*NO*]PARALLEL [*int*]

[*NO*]PARALLEL determines whether parallel recovery of media is used. *NOPARALLEL* is the default and enforces serial reading of the media. *PARALLEL* with no *int* value tells Oracle to choose the degree of parallelism to apply. Specifying *int* declares the degree of parallelism to apply.

RENAME GLOBAL_NAME TO *database*[.*domain*[.*domain* ...]]

Changes the global name of the database, where *database* is the new name of up to 8 bytes in length. The optional *domain* specifications identify the database's location in the network. This does not propagate database name changes to any dependent objects like synonyms, stored procedures, etc.

{DISABLE | ENABLE} BLOCK CHANGE TRACKING [USING FILE '*file*'] [REUSE]

Tells Oracle to stop or start tracking the physical location of all database updates, respectively, and maintain the information in a special file called the block change tracking file. Oracle will automatically create the file as defined by the *DB_CREATE_FILE_DEST* parameter, unless you add the *USING FILE 'file'* clause, where '*file*' is the path and name of the file. *REUSE* tells Oracle to overwrite an existing block change tracking file of the same name as '*file*'. The *USING* and *REUSE* subclauses are only allowed with the *ENABLE BLOCK* clause.

FLASHBACK {ON | OFF}

Places the database in or out of *FLASHBACK* mode, respectively. When in flashback mode, an Oracle database automatically creates and maintains flashback database logs in the flash recovery area. When *OFF*, the flashback database logs are deleted and unavailable.

SET TIME_ZONE

Specifies the time zone for the server. Refer to the *SET TIME_ZONE* statement for more information.

ENABLE [PUBLIC] THREAD *int* | DISABLE THREAD *int* | INSTANCE '*instance_name*'

In Real Application Cluster (RAC) environments, you can *ENABLE* or *DISABLE* a redo log thread by number (*int*). You may optionally specify an *instance_name* to enable or disable a thread mapped to a specific database instance of an Oracle Real Application Cluster environment. The *instance_name* may be up to 80 characters long. The *PUBLIC* keyword makes the thread available to any instance. When omitted, the thread is only available when explicitly requested. To enable a thread, the thread must have at least two redo logfile groups. To disable a thread, the database must be open but not mounted by an instance using the thread.

GUARD {ALL | STANDBY | NONE}

Protects the data in a database from changes. *ALL* prevents users other than SYS from making changes. *STANDBY* prevents all users other than SYS from making changes in a logical standby. *NONE* provides normal security for the database.

CREATE DATAFILE '*file*' [,...] [AS {NEW | *file_definition*}]

Creates a new empty datafile, replacing an existing one. The value '*file*' identifies a file (either by filename or file number) that was lost or damaged without a backup. *AS NEW* creates a new file in the default filesystem using an Oracle-supplied name. *AS file_definition* allows you to specify a filename and sizing details, as defined under the *file_definition* section earlier in the list.

DATAFILE '*file*' | TEMPFILE '*file*' [,...] {ONLINE | OFFLINE [FOR DROP]}
| RESIZE *int* [K | M] | AUTOEXTEND {OFF | ON [NEXT
int [K | M]]} [MAXSIZE [UNLIMITED | *int* [K | M]]} {END BACKUP [INCLUDING DATAFILES]}

Changes the attributes, such as the size, of one or more existing datafiles or tempfiles. You may alter one or more files in a comma-delimited list, identified in the value '*file*' by filename or file number. Do not mix datafile and tempfile declarations; only one or the other should appear in this clause at a time, where:

ONLINE

Sets the file online.

OFFLINE [FOR DROP]

Sets the file offline allowing media recovery. *FOR DROP* is required to take a file offline in *NOARCHIVELOG* mode, but does not actually destroy the file. It is ignored in *ARCHIVELOG* mode.

RESIZE *int* [K | M]

Sets a new size for an existing datafile or tempfile.

END BACKUP

Described later in the list under *END BACKUP*. Used only with the *DATAFILE* clause.

DROP INCLUDING DATAFILES

Drops not only the tempfile, but all datafiles on the filesystem associated with the tempfile. Oracle also adds an entry to the alert log for each file that is erased. Used only with the *TEMPFILE* clause.

AUTOEXTEND {OFF | ON [NEXT *int* [K | M] [MAXSIZE [UNLIMITED | *int* [K | M]]}

Described earlier in the list under *DATAFILE* definition.

RENAME FILE '*file*' [,...] TO '*new_file_name*' [,...]

Renames a datafile, tempfile, or redo logfile member from the old name, '*file*', to the '*new_file_name*'. You can rename multiple files at once by specifying multiple old and new filenames. This command does not rename files at the operating system level. Rather, it specifies new names that Oracle will use to open the files. You need to rename at the operating system level yourself.

[NO] FORCE LOGGING

Puts the database into force logging mode (*FORCE LOGGING*) or takes it out of force logging mode (*NO FORCE LOGGING*). In the former, Oracle logs all changes to the database except in temporary tablespaces or segments. This database-level *FORCE LOGGING* setting supersedes all tablespace level declarations regarding force logging mode.

[NO]ARCHIVELOG [MANUAL]

Tells Oracle to create redo logfiles, but the user will handle the archiving of the redo logfiles explicitly. It is used only with the *ALTER DATABASE* statement and only for backward compatibility for users with older tape backup systems. When this clause is omitted, Oracle defaults the redo logfile destination to the *LOG_ARCHIVE_DEST_n* initialization parameter.

[ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY) [,...] COLUMNS]

ADD places additional column data into the log stream whenever an update is executed. It also enables *minimal supplemental logging*, which ensures that LogMiner can support chained rows and special storage arrangements like clustered tables. Supplemental logging is disabled by default. You can add the clauses *PRIMARY KEY COLUMNS*, *UNIQUE KEY COLUMNS*, *FOREIGN KEY COLUMNS*, or *ALL* (to get all three earlier options), if you need to enable full referential integrity via foreign keys in another database, such as a logical standby. In either case, Oracle places either the primary key columns, unique key columns (or where none exist, a combination of columns that uniquely identify each row), foreign key columns, or all three into the log. *DROP* tells Oracle to suspend supplemental logging.

[ADD | DROP] [STANDBY] LOGFILE [THREAD *int*] [GROUP *int*] [MEMBER] '*file*' [,...]

ADD adds one or more primary or standby redo logfile groups to the specified instance. *THREAD* assigns the added files to a specific thread number (*int*) on a Real Application Cluster. When omitted, the default is the thread assigned to the current instance. *GROUP* assigns the redo logfile groups to a specific group within the thread. *MEMBER* adds the specified '*file*' (or files in a comma-delimited list) to an existing redo logfile group. *DROP LOGFILE MEMBER* drops one or more redo logfile members, after issuing an *ALTER SYSTEM SWITCH LOGFILE* statement.

CLEAR [UNARCHIVED] LOGFILE GROUP *int logfile_name* [,...] [UNRECOVERABLE DATAFILE] }

Reinitializes one or more specified online redo logs. Specify the logs in a comma-delimited list. *UNARCHIVED* skips archiving the redo logs. *UNRECOVERABLE DATAFILE* is required when any datafile is offline and the

database is in *ARCHIVELOG* mode.

CREATE {LOGICAL | PHYSICAL} STANDBY CONTROLFILE AS '*file*' [REUSE]}

Creates a control file that maintains a logical or physical standby database. *REUSE* is needed if the file already exists.

BACKUP CONTROLFILE TO {'*file*' [REUSE] | TRACE [AS '*file*' [REUSE]] [RESETLOGS | NORESETLOGS]}

Backs up the current control file of an open or mounted database. *TO 'file'* identifies a full path and filename for the control file. *TO TRACE* writes SQL statements to recreate the control file to a trace file. *TO TRACE AS 'file'* writes all the SQL statements to a standard file rather than a trace file. *REUSE* is needed if the file already exists. *RESETLOGS* initializes the trace file with the statement *ALTER DATABASE OPEN RESETLOGS*, and is only valid when online logs are unavailable. *NORESETLOGS* initializes the trace file with the statement *ALTER DATABASE OPEN NORESETLOGS* and is only valid when online logs are available.

RECOVER

Controls media recovery for the database, standby database, tablespace, or file. In Oracle, the *ALTER TABLE* command is one of the primary means of recovering a damaged or disabled database, file, or tablespace. Use *RECOVER* when the database is mounted (in exclusive mode), the files and tablespaces involved are not in use (offline), and the database is in either an open or closed state. The entire database can be recovered only when it is closed, but specific files or tablespaces can be recovered in a database that is open.

AUTOMATIC [FROM '*location*']

Tells Oracle to automatically generate the name of the next archived redo logfile necessary for continued operation during recovery. Oracle will prompt you if it cannot find the next file. The *FROM 'location'* clause tells Oracle where to find the archived redo logfile group. The following subclauses may be applied to an automatically recovered database.

STANDBY

Specifies that the database or tablespace to recover is a standby type.

DATABASE {UNTIL {CANCEL | TIME *date* | CHANGE *int*} | USING BACKUP CONTROLFILE }

Tells Oracle to recover the entire database. The *UNTIL* keyword tells Oracle to continue recovery until *ALTER DATABASE...RECOVER CANCEL (CANCEL)* is issued, until a specified time is reached (*TIME*) in the format YYYY-MM-DD:HH24:MI:SS, or until a specific system change number (*CHANGE int*) is reached where *int* is the number. The clause *USING BACKUP CONTROLFILE* enables use of the backup, rather than current, control file.

[STANDBY] TABLESPACE *tablespace_name* [,...] | DATAFILE '*file*' [,...]

Recovers one or more specific tablespaces or datafiles, respectively. You may specify more than one tablespace or datafile using a comma-delimited list. You may also recover a datafile by datafile number, rather than by name. The tablespace may be in normal or standby mode. The standby tablespace or datafile is reconstructed using archived redo logfiles copied from the primary database and a control file.

UNTIL [CONSISTENT WITH] CONTROLFILE

Tells Oracle to recover an older standby tablespace or datafile by using the current standby control file. *CONSISTENT WITH* are noise words.

LOGFILE *filename* [,...]

Continues media recovery by applying one or more redo logfiles that you specify in a comma-delimited list.

TEST | ALLOW *int* CORRUPTION | [NO]PARALLEL *int*

TEST performs a trial recovery, allowing you to foresee any problems. *ALLOW int CORRUPTION* tells how many corrupt blocks (*int*) to tolerate before causing recovery to abort. *int* must be 1 for a real recovery, but may be any number you chose when paired with *TEST*. Parallelism is described earlier in the list.

CONTINUE [DEFAULT]

Determines whether multi-instance recovery continues after interruption. *CONTINUE DEFAULT* is the same as *RECOVER AUTOMATIC*, but does not result in a prompt for a filename.

CANCEL [IMMEDIATE][[NO]WAIT]

Cancels a managed recovery operation at the next archived log boundary, if it was started with the *USING CANCEL* clause. The *IMMEDIATE* keyword cancels the managed recovery in process after the current I/O or archived log boundary, whichever is first. *NOWAIT* cancels the recovery immediately without regard to current work in process, while *WAIT* allows current work to complete normally.

MANAGED STANDBY DATABASE

Specifies managed physical standby recovery mode on an active component of a standby database. This command is used for media recovery only and not to construct a new database, using the following parameters:

[NO]TIMEOUT int

TIMEOUT specifies, in minutes (*int*), how long the foreground managed recovery operation will wait for the next available redo log before timing out. *NOTIMEOUT*, the default, tells Oracle to wait indefinitely for the next redo log, or until the instance is shutdown, or until *RECOVER CANCEL* is issued.

CANCEL [IMMEDIATE][[NO]WAIT]

Described earlier.

[NO]DELAY int | DEFAULT DELAY int

Tells, in minutes (*int*), how long Oracle should wait to apply an archived redo log after it has been selected for recovery. *DEFAULT DELAY* tells Oracle to use the delay specified by the *LOG_ARCHIVE_DEST_n INIT.ORA* parameter. *NODELAY* tells Oracle to apply the archived redo log immediately.

NEXT int

Tells Oracle to apply the *int* next archived redo logs as soon as possible after they have been archived. Incompatible with *FINISH*.

[NO]EXPIRE int

Tells Oracle, in minutes (*int*), when to automatically terminate a managed recovery relative to the time the recovery started. *NOEXPIRE* disables a previously set *EXPIRE* value. Incompatible with *FINISH*.

USING CURRENT LOGFILE

Invokes *real time apply*, which allows recovery of redos from standby online logs as they are being filled, without first requiring that they be archived by the standby database.

UNTIL CHANGE int

Conducts a managed recovery up to (but not including) the specified system change number called *int*.

THROUGH {ALL ARCHIVELOG | {ALL | LAST | NEXT} SWITCHOVER | [THREAD int] SEQUENCE int}

Tells Oracle when to terminate managed recovery to a standby server. *THROUGH ALL ARCHIVELOG*, the default, continues managed recovery until all archive logs are recovered. *THROUGH...SWITCHOVER* controls the termination of managed recovery when Oracle switches over to a secondary standby server. *ALL* keeps managed recovery in operation through any switchovers. *LAST* terminates managed recovery after receiving the final end-of-redo indicator. *NEXT* terminates managed recovery after

receiving the next end-of-redo indicator. *THROUGH...SEQUENCE* tells Oracle to terminate managed recovery when it encounters a specific *THREAD* number and/or *SEQUENCE* number.

DISCONNECT [FROM SESSION] [[NO]PARALLEL *int*]

Causes the managed redo process to occur in the background, leaving the current process available for other tasks. *FROM SESSION* are noise words. *DISCONNECT* is incompatible with *TIMEOUT*.

FINISH [SKIP [STANDBY LOGFILE]] [{WAIT | NOWAIT}]

Recovers the current physical standby online redo logfiles in preparation of the standby assuming the primary database role when the primary database has failed. The *FINISH* clause, when used without any other keywords, is known as a *terminal recovery*. The optional clause *SKIP [STANDBY LOGFILE]* allows the standby to prepare for the primary role when standby online redo logfiles are not available. The *FINISH SKIP* clause, when used without any other keywords, is known as a *terminal incomplete recovery*. *WAIT* tells Oracle to return control after completing the recovery process, while *NOWAIT* returns control immediately. Incompatible with *TIMEOUT*, *DELAY*, *EXPIRE*, and *NEXT*.

{*BEGIN* | *END*} *BACKUP*

Controls online backup mode for any datafiles. *BEGIN* places all datafiles into online backup mode. The database must be mounted and open, in archivelog mode with media recovery enabled. (Note that while the database is in online backup mode the instance cannot be shutdown, and individual tablespaces cannot be backed up, taken offline, or made read only.) *END* takes all datafiles currently in online backup mode out of that mode. The database must be mounted, but need not be open.

After that long discussion of specific syntax, it's important to establish some Oracle basics.

Oracle allows the use of *primary* and *standby* databases. A *primary* database is a mounted and open database accessible to users. The primary database then ships its redo logs to a *standby* database where they are recovered, thus making the standby database a very up-to-date copy of the primary.

Unique to the Oracle environment is the *INIT.ORA* file, which specifies the database name and a variety of other options when creating and starting up the database. You should always define startup parameters, such as the name of any control files, in the *INIT.ORA* file to identify the control files or else the database will not start. Starting in Oracle 9.1, you can use binary parameter files rather than *INIT.ORA* files.

When a group of logfiles is listed, they are usually shown in parentheses. The parentheses aren't needed when creating a group with only one member, but this is seldom done. Here's an example using a parenthetical list of logfiles:

```
CREATE DATABASE publications
```

```
LOGFILE ('/s01/oradata/loga01','/s01/oradata/loga02') SIZE 5M
```

```
DATAFILE;
```

The example above creates a database called **publications** with an explicitly defined logfile clause and an automatically created datafile. The following example of an Oracle *CREATE DATABASE* command is much more sophisticated:

```
CREATE DATABASE sales_reporting
```

```
CONTROLFILE REUSE
```

```
LOGFILE
```

```
GROUP 1 ('diskE:log01.log', 'diskF:log01.log') SIZE 15M,
```

```
GROUP 2 ('diskE:log02.log', 'diskF:log02.log') SIZE 15M
```

```
MAXLOGFILES 5
```

```
MAXLOGHISTORY 100
```

```
MAXDATAFILES 10  
MAXINSTANCES 2  
ARCHIVELOG  
CHARACTER SET AL32UTF8  
NATIONAL CHARACTER SET AL16UTF16  
DATAFILE  
  'diskE:sales_rpt1.dbf' AUTOEXTEND ON,  
  'diskF:sales_rpt2.dbf' AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED  
DEFAULT TEMPORARY TABLESPACE temp_tblspc  
UNDO TABLESPACE undo_tblspc  
SET TIME_ZONE = '-08:00';
```

This example defines log and data files, as well as all appropriate character sets. We also define a few characteristics for the database, such as the use of *ARCHIVELOG* mode and *CONTROLFILE REUSE* mode, the time zone, maximum number of instances, datafiles, etc. The example also assumes that the *INIT.ORA* parameter for *DB_CREATE_FILE_DEST* has already been set. Thus, we don't have to define file definitions for the *DEFAULT TEMPORARY TABLESPACE* and *UNDO TABLESPACE* clauses.

When issued by a user with SYSDBA privileges, this statement creates a database and makes it available to users in either exclusive or parallel mode, as defined by the value of the *CLUSTER_DATABASE* initialization parameter. Any data that exists in predefined datafiles is erased. You will usually want to create tablespaces and rollback segments for the database. (Refer to the vendor documentation for the platform specific commands *CREATE TABLESPACE* and *CREATE ROLLBACK SEGMENT*.)

Oracle has tightened up security around default database user accounts. Many default database user accounts are now locked and expired during initial installation. Only *SYS*, *SYSTEM*, *SCOTT*, *DBSNMP*, *OUTLN*, *AURORA\$JIS\$UTILITY\$*, *AURORA\$ORB\$UNAUTHENTICATED*, and *OSE\$HTTP\$ADMIN* are the same as they were in earlier versions. You must manually unlock and assign a new password to all locked accounts, as well as assign a password to *SYS* and *SYSTEM* during the initial installation.

In the next example, we add more logfiles to the current database, then add a datafile:

```
ALTER DATABASE ADD LOGFILE GROUP 3  
  ('diskf: log3.sales_arch_log','diskg:log3.sales_arch_log')  
SIZE 50M;  
ALTER DATABASE sales_archive  
CREATE DATAFILE 'diskF:sales_rpt4.dbf'  
AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED;
```

We can set a new default temporary tablespace, as shown in the next example:

```
ALTER DATABASE DEFAULT TEMPORARY TABLESPACE sales_tbl_spc_2;
```

Next, we'll perform a simple full database recovery:

```
ALTER DATABASE sales_archive RECOVER AUTOMATIC DATABASE;
```

In the next example, we perform a more elaborate partial database recovery:

```
ALTER DATABASE RECOVER STANDBY DATAFILE 'diskF:sales_rpt4.dbf'  
UNTIL CONTROLFILE;
```

Now, we'll perform a simple recovery of a standby database in managed standby recovery mode:

```
ALTER DATABASE RECOVER sales_archive MANAGED STANDBY DATABASE;
```

In the following example, we gracefully switch over from a primary database to a logical standby, and promote the logical standby to primary:

```
-- Demotes the current primary to logical standby database.
```

```
ALTER DATABASE COMMIT TO SWITCHOVER TO LOGICAL STANDBY;
```

```
-- Applies changes to the new standby.
```

```
ALTER DATABASE START LOGICAL STANDBY APPLY;
```

```
-- Promotes the current standby to primary database.
```

```
ALTER DATABASE COMMIT TO SWITCHOVER TO PRIMARY;
```

PostgreSQL

PostgreSQL's implementation of the *CREATE DATABASE* command creates a database and a file location for the datafiles:

```
CREATE DATABASE database_name  
  [ WITH {[LOCATION = 'dbpath']} [TEMPLATE = tmp_name]  
  [ENCODING = enc_value]} ]
```

Where:

*WITH LOCATION = '*dbpath*'*

Specifies the file path where the database will be located. You can leave off this clause to accept the default location or use the clause *WITH LOCATION = DEFAULT*. The path must be prepared in advance using the *initlocation* command (which is a Unix command, not a SQL statement). In general, PostgreSQL uses environmental variable names (an environmental variable name is a pathname without slashes). If you want to specify an absolute file and path name, like */usr/local/pgsql/data01*, you must set and compile the server with the *ALLOW_ABSOLUTE_DBPATH* parameter.

TEMPLATE = tmp_name

Names a template to use for creating the new database. You can leave off this clause to accept the default template describing the location of the database files or use the clause *TEMPLATE = DEFAULT*. The default is to copy the standard system database *template1*. You can get a pristine database (one that contains only required database objects) by specifying *TEMPLATE = template0*.

ENCODING = enc_value

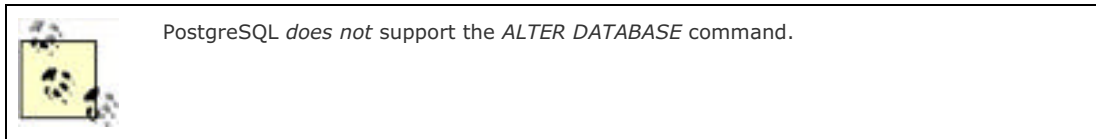
Specifies the multibyte encoding method to use in the new database using either a string literal (such as *'SQL_ASCII'*), an integer encoding number, or *DEFAULT* for the default encoding.

For example, to create the database **sales_revenue** in the */home/teddy/private_db* directory:

```
CREATE DATABASE sales_revenue WITH LOCATION =  
'/home/teddy/private_db';
```

Be careful using absolute file- and pathnames, since they impact security and data integrity issues.

PostgreSQL advises that databases used as templates be treated as read-only. Template databases should not be used for general purpose "copy database" functionality.



SQL Server

SQL Server offers a high degree of control over the OS filesystem structures that hold the database and its objects. The syntax for Microsoft SQL Server's *CREATE DATABASE* statement looks like this:

```
CREATE DATABASE database_name  
  
[ ON file_definition [...] ]  
  
[ , FILEGROUP filegroup_name file_definition [...] ]  
  
[ LOG ON file_definition [...] ]  
  
[ COLLATE collation_name ]  
  
[ FOR LOAD | FOR ATTACH ]
```

Following is the syntax for *ALTER DATABASE*:

```
ALTER DATABASE database_name  
  
{ADD FILE file_definition [...] [TO FILEGROUP filegroup_name]  
| ADD LOG FILE file_definition [...]  
| REMOVE FILE file_name  
| ADD FILEGROUP filegroup_name  
| REMOVE FILEGROUP filegroup_name  
| MODIFY FILE file_definition  
| MODIFY NAME = new_database_name  
| MODIFY FILEGROUP filegroup_name {filegroup_property  
| NAME = new_filegroup_name }  
| SET {state_option | cursor_option  
| auto_option | sql_option |  
recovery_option}  
[,...] [ WITH termination_option ]  
| COLLATE collation_name }
```

Parameter descriptions are as follows:

```
{CREATE | ALTER} DATABASE database_name
```

Creates a database, or alters an existing database, of *database_name*. The name cannot be longer than 128 characters. You should limit the database name to 123 characters when no logical filename is supplied, since SQL Server will create a logical filename by appending a suffix to the database name.

{ON | ADD} *file_definition* [,...]

Defines the disk file(s) that store the data components of the database for *CREATE DATABASE* or adds disk file(s) for *ALTER DATABASE*. *ON* is required for *CREATE DATABASE* only if you wish to provide one or more file definitions, where the syntax for *file_definition* is:

```
{[PRIMARY] ( [NEW][NAME = file_name] [, FILENAME = 'os_file_name']  
[, SIZE = int[KB | MD | GB | TB] ] [, MAXSIZE = { int |  
UNLIMITED } ] [, FILEGROWTH = int] ) } [,...]
```

where:

PRIMARY

Defines the *file_definition* as the primary file. Only one is allowed per database. (If you don't define a primary file, SQL Server defaults primary status to the file that it autcreates, in the absence of any user-defined file, or to the first file that you define.) The primary file, or groups of files (also called filegroups), contains the logical start of the database, all the database system tables, and all other objects not contained in user filegroups.

[NEW]NAME = *file_name*

Defines the logical name of the file defined by the *file_definition* for *CREATE DATABASE*. Use *NEWNAME* for *ALTER DATABASE* to define a new logical name for the file. In either case, the logical name must be unique within the database. This clause is optional when using *FOR ATTACH*.

FILENAME = '*os_file_name*'

Defines the operating system path and filename for the file defined by *file_definition*. The file must be in a non-compressed directory on the filesystem. For raw partitions, specify only the drive letter of the raw partition.

SIZE = *int*[KB | MB | GB | TB]

Defines the size of the file defined by the *file_definition*. This clause is optional, but defaults to the file size for the primary file of the model database, which is usually very small. Logfiles and secondary datafiles default to 1MB in size. The value of *int* defaults to megabytes; however, you can explicitly define the size of the file using suffixes for kilobyte (KB), megabyte (MB), gigabyte (GB), and terabyte (TB). Size cannot be smaller than 512KB or the size of the primary file of the model database.

MAXSIZE = { *int* | UNLIMITED }

Defines the maximum size the file may grow to. Suffixes, as described under the entry for *SIZE*, are allowed. The default, *UNLIMITED*, allows the file to grow until all available disk space is consumed. Not required for files on raw partitions.

FILEGROWTH = *int*

Defines the growth increment for the file each time it grows. Suffixes, as described under the entry for *SIZE*, are allowed. You may also use the percentage (%) suffix to indicate that the file should grow by the percentage of the total disk space currently consumed. If you omit the *FILEGROWTH* clause, the file will grow in 10% increments, but never less than 64KB. Not required for files on raw partitions.

[ADD] LOG {ON | FILE} *file_definition*

Defines the disk file(s) that store the log component of the database for *CREATE DATABASE* or adds disk file(s) for *ALTER DATABASE*. You can provide one or more *file_definitions* for the transaction logs in a comma-delimited list. Refer to the earlier section under the keyword *ON* for the full syntax of *file_definition*.

REMOVE FILE *file_name*

Removes a file from the database and deletes the physical file. The file must be emptied of all content first.

[ADD] FILEGROUP *filegroup_name file_definition* [,...]

Defines any user filegroups and their file definitions used by the database. All databases have at least one primary filegroup (though many databases only use the primary filegroup that comes with SQL Server by default). Adding filegroups and then moving files to the filegroups can allow you greater control over disk I/O. (We recommend that you do not add filegroups without careful analysis and testing.)

REMOVE FILEGROUP *filegroup_name*

Removes a filegroup from the database and deletes all the files in the filegroup. The files and the filegroup must be empty first.

MODIFY FILE *file_definition*

Changes the definition of a file. This clause is very similar to the {*ADD* | *ON*} *FILE* clause. For example: *MODIFY FILE* (*NAME* = *file_name*, *NEWNAME* = *new_file_name*, *SIZE*=...).

MODIFY NAME = *new_database_name*

Changes the database name from its current name to *new_database_name*.

MODIFY FILEGROUP *filegroup_name* {*filegroup_property* | NAME = *new_filegroup_name* }

Used with *ALTER DATABASE*, this clause has two forms. The first form allows you to change a filegroup name, as in *MODIFY FILEGROUP filegroup_name NAME = new_filegroup_name*. The second form allows you to specify a *filegroup_property* for the filegroup, which must be one of the following:

READONLY

Sets the filegroup to read-only and disallows updates to all objects within the filegroup. *READONLY* can only be enabled by users with exclusive database access and cannot be applied to the primary filegroup.

READWRITE

Disables the *READONLY* property and allows updates to objects within the filegroup. *READWRITE* can only be enabled by users with exclusive database access.

DEFAULT

Sets the filegroup as the default filegroup for the database. All new tables and indexes are assigned to the default filegroup unless explicitly assigned elsewhere. Only one default filegroup is allowed per database. (By default, the *CREATE DATABASE* statement sets the primary filegroup as the default filegroup.)

SET {*state_option* | *cursor_option* | *auto_option* | *sql_option* | *recovery_option*} [,...]

Controls a wide variety of behaviors for the database. These are discussed in the rules and information later in this section.

WITH *termination_option*

Used after the *SET* clause, *WITH* sets the rollback behavior for incomplete transactions whenever the database is in transition. When omitted, transactions must commit or rollback on their own with the database state changes. There are two *termination_option* settings:

ROLLBACK AFTER *int* [SECONDS] | ROLLBACK IMMEDIATE

Causes the database to roll back in *int* number of seconds, or immediately. *SECONDS* is a noise word and does not change the behavior of the *ROLLBACK AFTER* clause.

NO_WAIT

Causes database state or option changes to fail if a change cannot be completed immediately, without waiting for the current transaction to independently commit or roll back.

COLLATE *collation_name*

Defines or alters the default collation used by the database. *Collation_name* can be either a SQL Server collation name or a Windows collation name. By default, all new databases receive the collation of the SQL Server instance. (You can execute the query *SELECT * FROM ::fn_helpcollations()* to see all the collation names available.) To change the collation of a database, you must be the only user in the database, no schema bound objects that depend on the current collation may exist in the database, and the collation change must not result in the duplication of any object names in the database.

FOR LOAD | FOR ATTACH

Places the database in special startup mode. *FOR LOAD* is supported only for backward compatibility with versions earlier than 7.0 and sets the database to a status of loading and "dbo use only." *FOR ATTACH* creates the database from a set of pre-existing operating system files, almost always database files created previously. Because of this, the new database must have the same code page and sort order as the previous database. You only need the *file_definition* of the first primary file or those files that have a different path from the last time the database was attached. In general, you should use the *sp_attach_db* system stored procedure over the *CREATE DATABASE FOR ATTACH* statement unless you need to specify more than 16 *file_definitions*.

The *CREATE DATABASE* command should be issued from the *master* system database. You can, in fact, issue the command *CREATE DATABASE database_name*, with no other clauses, to get a very small, default database.

SQL Server uses *files*, formerly called *devices*, to act as a repository for databases. Files are grouped into one or more filegroups, with at least a *PRIMARY* filegroup assigned to each database. A file is a predefined block of space created on the disk structure. A database may be stored on one or more files or filegroups. SQL Server also allows the transaction log to be placed in a separate location from the database using the *LOG ON* clause. These functions allow sophisticated file planning for optimal control of disk I/O. For example, we can create a database called **sales_report** with a data and transaction logfile:

USE master

GO

CREATE DATABASE sales_report

ON

(NAME = sales_rpt_data, FILENAME =

'c:\mssql\data\salerptdata.mdf',

SIZE = 100, MAXSIZE = 500, FILEGROWTH = 25)

LOG ON

(NAME = 'sales_rpt_log',

FILENAME = 'c:\mssql\log\salesrptlog.ldf',

SIZE = 25MB, MAXSIZE = 50MB,

FILEGROWTH = 5MB)

GO

When a database is created, all objects in the model database are copied into the new database. All of the empty space within the file or files defined for the database is then initialized (i.e., emptied out), which means that creating a new and very large database can take a while, especially on a slow disk.

A database always has at least a primary datafile and a transaction logfile, but may also have secondary files for both the data and log components of the database. SQL Server uses default filename extensions: *.mdf* for primary datafiles, *.ndf* for secondary files, and *.ldf* for transaction logfiles. The following example creates a database called **sales_archive**

with several very large files that are grouped into a couple of filegroups:

```
USE master
GO
CREATE DATABASE sales_archive
ON
PRIMARY (NAME = sales_arch1, FILENAME = 'c:\mssql\data\archdata1.mdf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sales_arch2,
        FILENAME = 'c:\mssql\data\archdata2.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sales_arch3,
        FILENAME = 'c:\mssql\data\archdat3.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB)
FILEGROUP sale_rpt_grp1
        (NAME = sale_rpt_grp1_1_data,
        FILENAME = 'c:\mssql\data\SG1Fi1dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sale_rpt_grp1_1_data,
        FILENAME = 'c:\mssql\data\SG1Fi2dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
FILEGROUP sale_rpt_grp2
        (NAME = sale_rpt_grp2_1_data, FILENAME = 'c:\mssql\data\SRG21dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
        (NAME = sale_rpt_grp2_2_data, FILENAME = 'c:\mssql\data\SRG22dt.ndf',
        SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
LOG ON
        (NAME = sales_archlog1,
        FILENAME = 'd:\mssql\log\archlog1.ldf',
        SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%),
        (NAME = sales_archlog2,
        FILENAME = 'd:\mssql\log\archlog2.ldf',
        SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%)
GO
```

The *FOR ATTACH* clause is commonly used for situations like a salesperson traveling with a database on a CD. This clause tells SQL Server that the database is attached from an existing operating-system file structure, such as a DVD-ROM, CD-ROM, or portable hard drive. When using *FOR ATTACH*, the new database inherits all the objects and data of the parent database, not the model database.

The following examples show how to change the name of a database, file, or filegroup:

```
-- Rename a database

ALTER DATABASE sales_archive MODIFY NAME = sales_history

GO

-- Rename a file

ALTER DATABASE sales_archive MODIFY FILE

NAME = sales_arch1,

NEWNAME = sales_hist1

GO

-- Rename a filegroup

ALTER DATABASE sales_archive MODIFY FILEGROUP

sale_rpt_grp1

NAME = sales_hist_grp1

GO
```

There may be times when you want to add new free space to a database, especially if you have not enabled it to auto-grow:

```
USE master

GO

ALTER DATABASE sales_report ADD FILE

( NAME = sales_rpt_added01, FILENAME = 'c:\mssql\data\salerptadded01.mdf',

  SIZE = 50MB, MAXSIZE = 250MB, FILEGROWTH = 25MB )

GO
```

When you alter a database, you can set many behavior options on the database. State options (shown as *state_option* in the earlier syntax diagram) control how users access the database. Following is a list of valid state options:

SINGLE_USER | *RESTRICTED_USER* | *MULTI_USER*

Sets the number and type of users with access to the database. *SINGLE_USER* mode allows only one user to access the database at a time. *RESTRICTED_USER* mode allows access only members of the system roles *db_owner*, *dbcreator*, or *sysadmin*. *MULTI_USER*, the default, allows concurrent database access from all users who have permission.

OFFLINE | *ONLINE*

Sets the database to offline (unavailable) or online (available).

READ_ONLY | *READ_WRITE*

Sets the database to *READ_ONLY* mode, where no modifications are allowed, or to *READ_WRITE* mode, where data modifications are allowed. *READ_ONLY* databases can be very fast for query intensive operations, since almost no locking is needed.

Cursor options control default behavior for cursors in the database. In the *ALTER DATABASE* syntax shown earlier, you can replace *cursor_option* with any of the following:

CURSOR_CLOSE_ON_COMMIT { *ON* | *OFF* }

When set *ON*, any open cursors are closed when a transaction commits or rolls back. When set *OFF*, any open

cursors remain open when transactions are committed and open cursors close when a transaction rolls back unless the cursor is *INSENSITIVE* or *STATIC*. (See [DECLARE CURSOR Command](#) later in this chapter for more details.)

CURSOR_DEFAULT { *LOCAL* | *GLOBAL* }

Sets the default scope of all cursors in the database to either *LOCAL* or *GLOBAL*. (See [DECLARE CURSOR Command](#) later in this chapter for more details.)

In the SET clause, *auto_option* controls the automatic file handling behaviors of the database. The following are valid replacements for *auto_option*:

AUTO_CLOSE { *ON* | *OFF* }

When set *ON*, the database automatically shuts down cleanly and frees all resources when the last user exits. When set *OFF*, the database remains open when the last user exits. The default is *OFF*.

AUTO_CREATE_STATISTICS { *ON* | *OFF* }

When set *ON*, statistics are automatically created when SQL Server notices they are missing during query optimization. When set *OFF*, statistics are not created during optimization. The default is *ON*.

AUTO_SHRINK { *ON* | *OFF* }

When set *ON*, the database files may automatically shrink. The database periodically looks for an opportunity to shrink files, though the time is not always predictable. When set *OFF*, files will shrink only when you explicitly and manually shrink them. The default is *OFF*.

AUTO_UPDATE_STATISTICS { *ON* | *OFF* }

When set *ON*, out-of-date statistics are reassessed during query optimization. When set *OFF*, statistics are reassessed only by explicitly and manually recompiling them using the SQL Server command *UPDATE STATISTICS*.

The *sql_options* clause controls ANSI compatibility of the database. You can use the standalone SQL Server command *SET ANSI_DEFAULTS ON* to enable all the ANSI SQL92 behaviors at one time, rather than using the individual statements below. In the SET clause, you can replace *sql_option* with any of the following:

ANSI_NULL_DEFAULT { *ON* | *OFF* }

When set *ON*, the *CREATE TABLE* statement causes columns with no nullability setting to default to NULL. When set *OFF*, the nullability of a column defaults to NOT NULL. The default is *OFF*.

ANSI_NULLS { *ON* | *OFF* }

When set *ON*, comparisons to null yield *UNKNOWN*. When set *OFF*, comparisons to null yield null if both non-UNICODE values are null. The default is *OFF*.

ANSI_PADDING { *ON* | *OFF* }

When set *ON*, strings are padded to the same length for insert or comparison operations on *VARCHAR* and *VARBINARY* columns. When set *OFF*, strings are not padded. The default is *ON*. (We recommend that you *do not* change this!)

ANSI_WARNINGS { *ON* | *OFF* }

When set *ON*, the database warns when problems like "divide by zero" or "null in aggregates" occur. When set *OFF*, these warnings are not raised. The default is *OFF*.

ARITHABORT { *ON* | *OFF* }

When set *ON*, divide-by-zero and overflow errors cause a query or Transact-SQL batch to terminate and roll back any open transactions. When set *OFF*, a warning is raised but processing continues. The default is *ON*. (We

recommend that you *do not* change this!)

CONCAT_NULL_YIELDS_NULL { *ON* | *OFF* }

When set *ON*, returns a null when a null is concatenated to a string. When set *OFF*, nulls are treated as empty strings when concatenated to a string. The default is *OFF*.

NUMERIC_ROUNDABORT { *ON* | *OFF* }

When set *ON*, an error is raised when a numeric expression loses precision. When set *OFF*, losses of precision result in rounding of the result from the numeric expression. The default is *OFF*.

QUOTED_IDENTIFIER { *ON* | *OFF* }

When set *ON*, double quotation marks identify an object identifier with special characters or one that is a reserved word (e.g., a table named **SELECT**). When set *OFF*, identifiers may not contain special characters or reserved words and all occurrences of double quotation marks signify a literal string value. The default is *OFF*.

RECURSIVE_TRIGGERS { *ON* | *OFF* }

When set *ON*, triggers can fire recursively. That is, the actions taken by one trigger may cause another trigger to fire, and so on. When set *OFF*, triggers cannot cause other triggers to fire. The default is *OFF*.

Recovery options control the recovery model used by the database. Use any of the following in place of *recovery_option* in the *ALTER DATABASE* syntax:

RECOVERY { *FULL* | *BULK_LOGGED* | *SIMPLE* }

When set to *FULL*, database backups and transaction logs provide full recoverability even for bulk operations like *SELECT...INTO*, *CREATE INDEX*, etc. *FULL* is the default for SQL Server 2000 Standard Edition and Enterprise Edition. *FULL* provides the most recoverability, even from a catastrophic media failure, but uses more space. When set to *BULK_LOGGED*, logging for bulk operations is minimized. Space is saved and fewer I/Os are incurred, but risk of data loss is greater than under *FULL*. When set to *SIMPLE*, the database can only be recovered to the last full or differential backup. *SIMPLE* is the default for SQL Server 2000 Desktop Edition and Personal Edition.

TORN_PAGE_DETECTION { *ON* | *OFF* }

When set *ON*, SQL Server can detect incomplete I/O operations at the disk level by checking each 512-byte sector per 8K database page. (Torn pages are usually detected in recovery.) The default is *ON*.

For example, we may want to change some behavior settings for the **sales_report** database without actually changing the underlying file structure:

```
ALTER DATABASE sales_report SET ONLINE, READ_ONLY,
```

```
AUTO_CREATE_STATISTICS ON
```

```
GO
```

This statement puts the database online and in read-only mode. It also sets the *AUTO_CREATE_STATISTICS* behavior to *ON*.

See Also

```
CREATE SCHEMA  
DROP STATEMENT
```

CREATE/ALTER FUNCTION/PROCEDURE Statements

The *CREATE FUNCTION* and *CREATE PROCEDURE* statements are very similar in syntax and coding (as are the respective *ALTER* statements).

The *CREATE PROCEDURE* statement creates a *stored procedure*, which takes input arguments and performs conditional processing against various objects in the database. According to the ANSI standard, a stored procedure returns no result set (though it may return a value in an *OUTPUT* parameter). For example, you might use a stored procedure to perform all the processes that close an accounting cycle.

The *CREATE FUNCTION* statement creates a *user-defined function* (UDF), which takes input arguments and returns a single value output in the same way as system-supplied functions like *CAST()* or *UPPER()*. These functions, once created, can be called in queries and data-manipulation operations, such as *INSERT*, *UPDATE*, and the *WHERE* clause of *DELETE* statements. Refer to [Chapter 4](#) for descriptions of built-in SQL functions and their individual vendor implementations.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with variations

SQL2003 Syntax

Use the following syntax to create a stored procedure or function:

```
CREATE {PROCEDURE | FUNCTION} object_name
  ( ( {[IN | OUT | INOUT] [parameter_name] datatype [AS LOCATOR] [RESULT]}
    [...]) )
  [ RETURNS datatype [AS LOCATOR]
    [CAST FROM datatype [AS LOCATOR] ]
  [LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
  [PARAMETER STYLE {SQL | GENERAL}]
  [SPECIFIC specific_name]
  [DETERMINISTIC | NOT DETERMINISTIC]
  [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
  [RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
  [DYNAMIC RESULT SETS int]
  [STATIC DISPATCH]
  code_block
```

Use the following syntax to alter a pre-existing UDF or stored procedure:

```
ALTER {PROCEDURE | FUNCTION} object_name
  [( {parameter_name datatype } [...])]
  [NAME new_object_name]
```

[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]

[PARAMETER STYLE {SQL | GENERAL}]

[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]

[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]

[DYNAMIC RESULT SETS *int*]

[CASCADE | RESTRICT]

Keywords

CREATE {PROCEDURE | FUNCTION} [AS LOCATOR] [RESULT] *object_name*

Creates a new stored procedure or user-defined function with the name *object_name*. A user-defined function returns a value, where a stored procedure, in ANSI SQL, does not.

[(IN | OUT | INOUT) [*parameter_name*] *datatype* [,...])

Declares one or more parameters to be passed into a stored procedure in a comma-delimited list enclosed in parentheses. Parameters used with a procedure may pass a value *IN*, *OUT*, or both in and out via *INOUT*.

The syntax for the parameter declaration is:

[{IN | OUT | INOUT}] *parameter_name1 datatype*,

[{IN | OUT | INOUT}] *parameter_name2 datatype*,[...]

To provide an optional *parameter_name*, make sure the name is unique within the stored procedure. The optional *AS LOCATOR* subclause is used to validate an external routine with a *RETURNS* parameter that is a *BLOB*, *CLOB*, *NCLOB*, *ARRAY*, or user-defined type. If you need to change the datatype of a *RETURNS* parameter on the fly, use the *CAST* clause (refer to the function, *CAST*, in [Chapter 4](#)). For example, *RETURNS VARCHAR(12) CAST FROM DATE*. When used with *ALTER*, this clause adds parameters to a pre-existing stored procedure. Refer to [Chapter 2](#) for details on datatypes.

RETURNS *datatype* [AS LOCATOR] [CAST FROM *datatype* [AS LOCATOR]]

Declares the datatype of the result returned by a function. (This clause is used only in the *CREATE FUNCTION* statement and is not used in stored procedures.) The key purpose of a user-defined function is to return a value.

LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}

Declares the language in which the function is written. Most database platforms do not support all of these languages, and often support several not mentioned, like Java. When omitted, the default is SQL. When used with *ALTER*, this clause changes the existing *LANGUAGE* value to the value that you declare.

PARAMETER STYLE {SQL | GENERAL}

Indicates, for external routines only, whether certain implicit and automatic parameters are passed, explicitly, with the options (*SQL*) or not (*GENERAL*). (The difference between SQL style and GENERAL style is that SQL style passes automatic SQL parameters, such as indicators, while the GENERAL style does not automatically pass parameters.) The default is *PARAMETER STYLE SQL*. When used with *ALTER*, this clause changes the existing *PARAMETER STYLE* value to the value that you declare.

SPECIFIC *specific_name*

Uniquely identifies the function generally used with user-defined types.

DETERMINISTIC | NOT DETERMINISTIC

States the nature of values returned by a function. (This clause is used only in *CREATE* and *ALTER FUNCTION* statement.) *DETERMINISTIC* functions always return the same value when given the same parameter values. *NOT DETERMINISTIC* functions may return variable results when given the same parameter values. For example, *CURRENT_TIME* is not deterministic because it returns a constantly advancing value corresponding to the time.

NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA

Specifies, in conjunction with the *LANGUAGE* setting, the type of SQL contained in the user-defined function. When used with *ALTER*, this clause changes the existing *SQL* style value to the value that you declare.

NO SQL

Indicates that no SQL statements of any type are in the function. It is used with a non-SQL *LANGUAGE* setting such as *LANGUAGE ADA...CONTAINS NO SQL*.

CONTAINS SQL

Indicates that SQL statements other than read or modify statements are in the function. This is the default.

READS SQL DATA

Indicates the function contains *SELECT* or *FETCH* statements.

MODIFIES SQL DATA

Indicates the function contains *INSERT*, *UPDATE*, or *DELETE* statements.

RETURN NULL ON NULL INPUT | CALL ON NULL INPUT

These options are for use with a host *LANGUAGE* that cannot support nulls. The *RETURNS NULL ON NULL INPUT* setting causes the function to immediately return a null value if it is passed a null value. The *CALL ON NULL INPUT* setting causes the function to handle nulls according to standard rules, for example returning UNKNOWN when a comparison of two null values occurs. (This clause is used in the *CREATE* and *ALTER PROCEDURE* and *FUNCTION* statements.) When used with *ALTER*, this clause changes the existing NULL style value to the value that you declare.

DYNAMIC RESULT SETS int

Declares that a certain number of cursors, *int*, can be opened by the stored procedure and that those cursors are visible after returning from the procedure. When omitted, the default is *DYNAMIC RESULT SETS 0*. (This clause is not used in *CREATE FUNCTION* statements.) When used with *ALTER*, this clause changes the existing *DYNAMIC RESULT SETS* value to the value that you declare.

STATIC DISPATCH

Returns the static values of a user-defined type or *ARRAY* datatype. Required for non-SQL functions that contain parameters that use user-defined types or *ARRAYS*. (This clause is not used in *CREATE PROCEDURE* statements.) This clause must be the last clause in the function or procedure declaration before the *code_block*.

code_block

Declares the procedural statements that handle all processing within the user-defined function or stored procedure. This is the most important, and usually largest, part of a function or procedure. We assume you're interested in a user-defined function of *LANGUAGE SQL*, so note that the *code_block* may not contain SQL-Schema statements, SQL-Transaction statements, nor SQL-Connection statements. While we assume that you're interested in SQL-based UDFs and stored procedures (this is a SQL book after all), you can declare that the code block is derived externally. The syntax for external *code_blocks* is:

EXTERNAL [**NAME** *external_routine_name*] [**PARAMETER STYLE**

{**SQL | GENERAL**}] [**TRANSFORM GROUP** *group_name*]

where:

EXTERNAL [NAME *external_routine_name*]

Defines an external routine and assigns a name to it. When omitted, the unqualified routine name is used.

PARAMETER STYLE [SQL | GENERAL]

Same as for *CREATE PROCEDURE*.

TRANSFORM GROUP *group_name*

Transforms values between user-defined types and host variables in a user-defined function or a stored procedure. When omitted, the default is *TRANSFORM GROUP DEFAULT*.

NAME *new_object_name*

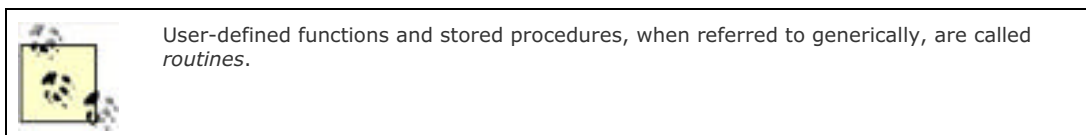
Declares the new name to use for a previously defined UDF or stored procedure. This clause is only used with *ALTER FUNCTION* and *ALTER PROCEDURE* statements.

CASCADE | RESTRICT

Allows you to cause changes to *CASCADE* down to all dependent UDFs or stored procedures, or to *RESTRICT* a change from happening if there are dependent objects. We strongly recommend that you *do not* issue an *ALTER* statement against UDFs or stored procedures that have dependent objects. This clause is only used with *ALTER FUNCTION* and *ALTER PROCEDURE* statements.

Rules at a Glance

With a user-defined function (UDF), you declare the input arguments and the single output argument that the function passes back out. You can then call the user-defined function just as you would any other function, for example, in *SELECT* statements, *INSERT* statements, or the *WHERE* clause of *DELETE* statements.



With a stored procedure, you declare the input arguments that go into the stored procedure and the output arguments that come out from it. You invoke a stored procedure using the *CALL* statement.

The content of the *code_block* must conform to the rules of whatever procedural language is supported by the database platform. Some platforms do not have their own internal procedural language, requiring you to use *EXTERNAL code_block* constructs.

For example, you might want build a user-defined function on Microsoft SQL Server that returns the first and last name of a person as a single string:

```
CREATE FUNCTION formatted_name (@fname VARCHAR(30),
```

```
@lname VARCHAR(30) )
```

```
RETURNS VARCHAR(60)
```

```
AS
```

```
BEGIN
```

```
    DECLARE @full_name VARCHAR(60)
```

```
SET @full_name = @fname + ' ' + @lname  
RETURN @full_name  
END
```

You could then use this user-defined function as any other function:

```
SELECT formatted_name(au_fname, au_lname) AS name, au_id AS id  
FROM authors
```

The *SELECT* statement in this example will return a result set with the two columns, **name** and **id**.



The ANSI statement *CREATE METHOD* is supported only by DB2 at this time. A method is essentially a special purpose user-defined function and follows the same syntax.

Stored procedures behave similarly. In the following small example, this Microsoft SQL Server stored procedure generates a unique 22-digit value (based on elements of the system date and time) and returns it to the calling process:

```
-- A Microsoft SQL Server stored procedure  
CREATE PROCEDURE get_next_nbr  
    @next_nbr CHAR(22) OUTPUT  
AS  
BEGIN  
    DECLARE @random_nbr INT  
    SELECT @random_nbr = RAND( ) * 1000000  
    SELECT @next_nbr =  
        RIGHT('000000' + CAST(ROUND(RAND(@random_nbr)*1000000,0))AS  
            CHAR(6), 6) +  
        RIGHT('0000' + CAST(DATEPART (yy, GETDATE( ) ) AS CHAR(4)), 2) +  
        RIGHT('000' + CAST(DATEPART (dy, GETDATE( ) ) AS CHAR(3)), 3) +  
        RIGHT('00' + CAST(DATEPART (hh, GETDATE( ) ) AS CHAR(2)), 2) +  
        RIGHT('00' + CAST(DATEPART (mi, GETDATE( ) ) AS CHAR(2)), 2) +  
        RIGHT('00' + CAST(DATEPART (ss, GETDATE( ) ) AS CHAR(2)), 2) +  
        RIGHT('000' + CAST(DATEPART (ms, GETDATE( ) ) AS CHAR(3)), 3)  
END  
GO
```

In this next, and final, ANSI SQL2003 example, we change the name of an existing stored procedure:

```
ALTER PROCEDURE get_next_nbr  
NAME 'get_next_ID'  
RESTRICT;
```

Programming Tips and Gotchas

A key advantage of a stored procedure or function is the fact that it is *precompiled*, meaning that once created its query plans are already stored in the database. Precompiled routines can often (though not always) be cached in database memory to provide an additional boost in performance. A stored procedure or user-defined function can perform many statements with a single communication to the server, thus reducing network traffic.

Implementations of user-defined functions and stored procedures vary widely by platform. Some database platforms do not support internal *code_block* content. On these platforms, you can only write an external *code_block*. Be sure to read about the variations and capabilities on each platform.

If you execute an *ALTER PROCEDURE* or *FUNCTION* statement, dependent objects may become invalid after a change to an object they depend on. Be careful to check all dependencies when altering UDFs or stored procedures on which other UDFs or stored procedures may depend.

DB2

DB2's implementations of these commands are very close to the ANSI standard. Although the syntax diagrams are long, much of the usage is identical to the ANSI standard. Consequently, the keyword explanation section that follows the syntax diagram is relatively short, since many of the clauses are explained in the SQL standard.

DB2 supports several classifications of functions. Each kind of UDF has its own set of allowable syntax, though much of it is shared across the various kinds of UDF. DB2 allows external scalar functions, external table functions, OLE-DB external table functions, sourced (or template) functions, and SQL functions. DB2 supports two kinds of stored procedures: external and SQL.

The following syntax is the full syntax to create any sort of stored procedure or a UDF in DB2. It is grouped together here to give you a concise list of clause definitions. However, refer to the sections below for greater detail on the specific allowable syntax for each type of UDF and stored procedure. If you prefer, for example, to see only the allowable syntax for a SQL user-defined function, then skip to the section below. The total available syntax for UDFs and stored procedures follows:

```
CREATE {PROCEDURE | FUNCTION} object_name

    [( {[IN | OUT] INOUT} [parameter_name] datatype } [...]])

RETURNS {datatype | {ROW | TABLE} column_list} [AS LOCATOR]

    [CAST FROM datatype [AS LOCATOR] ]

[LANGUAGE {C | JAVA | COBOL | OLE | SQL}]

[PARAMETER STYLE {SQL | DB2GENERAL | GENERAL | GENERAL WITH NULLS | JAVA}]

[SPECIFIC specific_name]

[EXTERNAL NAME external_routine_name]

[DETERMINISTIC | NOT DETERMINISTIC]

[ [NO] EXTERNAL ACTION]

[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]

[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]

[DYNAMIC RESULT SETS int]

[STATIC DISPATCH]

[SOURCE object_name | AS TEMPLATE ]

[INHERIT SPECIAL REGISTERS]

[[NOT] FENCED] [[NOT] THREADSAFE]

[PROGRAM TYPE {SUB | MAIN}]

[ [NO] SCRATCHPAD int] [ [NO] FINAL CALL]
```

[{ALLOW | DISALLOW} PARALLEL]

[[NO] DBINFO]

[TRANSFORM GROUP *group_name*]

[CARDINALITY *int*]

[PREDICATES (*predicate_definition*)]

code_block

In addition, use the following syntax to alter a pre-existing UDF or stored procedure, and assign it the characteristics that you declare:

ALTER {PROCEDURE | FUNCTION} *object_name*

[EXTERNAL NAME *external_routine_name*]

[[NOT] FENCED] [[NOT] THREADSAFE]

Following are the parameter descriptions for DB2:

RETURNS {*datatype* | {ROW | TABLE} *column_list*}

Allows you to return a single value of *datatype*, a ROW of values, or a TABLE of values where *column_list* uses the syntax *column_name1 datatype1, column_name2 datatype2 [...]*.

LANGUAGE {C | JAVA | COBOL | OLE | SQL}

Same as the ANSI standard, though DB2 supports other languages than in the ANSI standard. When you declare a LANGUAGE setting, other options are restricted or made available.

PARAMETER STYLE {SQL | DB2GENERAL | GENERAL | GENERAL WITH NULLS | JAVA}

Similar to the ANSI SQL standard but with a richer variety of choices. PARAMETER STYLE is only used with external stored procedures or user-defined functions. The various styles determine how parameters are passed, where:

SQL

Defines that when parameters are passed with the CALL statement, a null indicator for each parameter and a qualified and specific name for the stored procedure or function are also passed. In addition, the SQLSTATE and SQL diagnostic string are passed back to DB2.

DB2GENERAL

Available only with the LANGAUGE JAVA clause, the DB2GENERAL clause specifies that the parameters are passed using the convention defined for Java.

GENERAL

Passes parameters directly from the CALL statement to the stored procedure without using a SQLDA structure. Nulls are not passed. Only available with LANGUAGE C or COBOL.

GENERAL WITH NULLS

Same as GENERAL, but also passes null indicators.

JAVA

Passes parameters using the Java and SQLJ Routines specifications. For example, *INOUT* and *OUT* parameters are passed as a single entry array. Available only with the *LANGUAGE JAVA* clause. This clause is mutually exclusive of the *DBINFO* clause and *PROGRAM TYPE* clause.

EXTERNAL NAME [*external_routine_name*]

Registers a new function or procedure using a linked and interfaced external program. When *external_routine_name* is omitted, *object_name* is assumed to be the name of the external program. The method of linking depends on the *LANGUAGE* setting.

NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA

Same as the ANSI standard, though certain options are not usable for certain *LANGUAGE* settings.

INHERIT SPECIAL REGISTERS

Declares that updateable special registers receive their initial value from the invoking process. The invoking process could be a UDF, a stored procedure, trigger, view, or user issuing a *CALL* statement.

[[NOT] FENCED] [{THREADSAFE | NOT THREADSAFE}]

Specifies whether (or *NOT*) the external stored procedure or user-defined function is "safe" to run in DB2's manager operating environment process or address space. This is used only for external stored procedures. *NOT FENCED* procedures tend to be faster, but open the database up to potential risks.

[[NOT] THREADSAFE]

Declares whether the method is safe to run in the same process as other routines (*THREADSAFE*) or not (*NOT THREADSAFE*). When declared *THREADSAFE*, a procedure is considered safe to run in the same process as other UDFs and stored procedures. *NOT THREADSAFE* can only be used with *FENCED* stored procedures.

PROGRAM TYPE {SUB | MAIN}

Tells whether the external stored procedure should expect parameters in the style of a *SUB* routine or a *MAIN* routine. *SUB*, the default, declares that the parameters are expected as separate arguments. *MAIN* declares that the parameters are expected as an argument counter and argument vector. *MAIN* also requires that the external stored procedure is named "main" and that it is built as a shared library, not as a standalone executable. *PROGRAM TYPE MAIN* is only valid with *LANGUAGE C* or *COBOL* and *PARAMETER STYLE GENERAL*, *GENERAL WITH NULLS*, *SQL*, or *DB2SQL*.

[NO] SCRATCHPAD *int*

Provides a scratch pad for any external function or procedure. A *SCRATCHPAD* provides a way for a UDF to save its state from one call to the next. The default is *NO SCRATCHPAD*. If an *int* is specified, it represents the size of the scratch pad in bytes (defaulting to 100 bytes). One scratch pad is initialized per reference to an external routine.

[NO] FINAL CALL

FINAL CALL causes the routine to invoke two extra times. The first invocation occurs before data is first loaded so the routine can perform any one-time initialization needed. The second invocation occurs last and allows the routine to do any cleanup needed. Table functions are always a *FINAL CALL*. This clause is incompatible with *PARAMETER STYLE JAVA*. The clause *NO FINAL CALL* prevents a final call from happening.

{ALLOW | DISALLOW} PARALLEL

Allows the routine to be split in parallel across multiple CPUs (a.k.a. parallelized) during plan optimization on an SMP or MPP server, with *ALLOW PARALLEL*. Otherwise, DB2 enforces the serial processing of a routine on a single CPU, regardless of how many CPUs exist on the server, with *DISALLOW PARALLEL*. *ALLOW PARALLEL* is the default, unless the routine contains any of these clauses: *NOT DETERMINISTIC*, *EXTERNAL ACTION*, *SCRATCHPAD*, or *FINAL CALL*.

[NO] DBINFO

Tells whether the external stored procedure receives an additional argument (*DBINFO*) or not (*NO DBINFO*, the default) when invoked. The *DBINFO* argument tells the database name, application ID, application authorization ID, code page, database release version, and platform. *DBINFO* is mutually exclusive of *LANGAUGE OLD* and *PARAMETER STYLE JAVA* and *DB2GENERAL*.

TRANSFORM GROUP *group_name*

Declares the transform group, *group_name*, if the routine uses a user-defined structured type for either a parameter or return datatype.

CARDINALITY *int*

PREDICATES (*predicate_definition*)

Used with *CREATE FUNCTION* to create a user-defined predicate. This option is used when defining user-defined functions for use with DB2's index extensions. Index extensions are quite complex and not used by the average DB2 user. Consequently, you should refer to DB2's documentation on index extensions for information on this clause. In addition, *DETERMINISTIC* with *NO EXTERNAL ACTION* clauses are required to use *PREDICATES*.

DB2 accommodates several kinds of user-defined functions: external scalar, external table, OLE-DB, source/template, and SQL, and two kinds of stored procedures: external and SQL. Each of these types allows a mix of only some of the clauses shown above.

DB2 external scalar user-defined functions

This variation of the *CREATE FUNCTION* statement is used to register a user-defined external scalar function with an application server. (Remember that *scalar functions* return a single value each time they are invoked and are valid wherever SQL expressions are valid.) An external scalar UDF allows the following clauses:

CREATE FUNCTION object_name (parameter_name datatype

[,...]) [AS LOCATOR]

RETURNS datatype [AS LOCATOR] [CAST FROM datatype

[AS LOCATOR]]

[LANGUAGE {C | JAVA | OLE}] [PARAMETER STYLE

{SQL | DB2GENERAL | JAVA}]

[SPECIFIC specific_name]

[EXTERNAL NAME external_routine_name]

[DETERMINISTIC | NOT DETERMINISTIC] [STATIC DISPATCH]

[[NO] EXTERNAL ACTION]

[NO SQL | CONTAINS SQL | READS SQL DATA]

[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]

[INHERIT SPECIAL REGISTERS]

[[NOT] FENCED [{THREADSAFE | NOT THREADSAFE}]]

[[NO] SCRATCHPAD int] [[NO] FINAL CALL]

[{ALLOW | DISALLOW} PARALLEL]

[[NO] DBINFO] [TRANSFORM GROUP group_name]

[PREDICATES (predicate_definition)]

code_block

The following code is an example of an external scalar UDF, called **report_reader.center**, that defines the center of a circle. The statement explicitly declares all option clauses, even those that have identical defaults to the declared values:

```
CREATE FUNCTION report_reader.center (FLOAT, FLOAT, FLOAT)
RETURNS DECIMAL(8,4) CAST FROM FLOAT
SPECIFIC center04
EXTERNAL NAME 'center!focalpt'
LANGUAGE C PARAMETER STYLE SQL DETERMINISTIC FENCED NOT NULL CALL
NO SQL NO EXTERNAL ACTION SCRATCHPAD NO FINAL CALL
DISALLOW PARALLEL
```

Note that the **report_reader.center** UDF uses a scratch pad and can accumulate and act on results stored there

DB2 external table user-defined functions

This variation of the *CREATE FUNCTION* statement is used to register a user-defined external table function with an application server. (A *table function* can be used in the *FROM* clause of a *SELECT* statement, returning a table to the *SELECT* one row at a time.) An external table UDF allows the following clauses:

```
CREATE FUNCTION object_name (parameter_name datatype }
    [AS LOCATOR] [...])
RETURNS TABLE datatype [AS LOCATOR] [...] ]
[LANGUAGE {C | JAVA | OLE} ] [PARAMETER STYLE
    {SQL | DB2GENERAL}]
[specific_name] [EXTERNAL NAME external_routine_name]
[DETERMINISTIC | NOT DETERMINISTIC] [ [NO] EXTERNAL ACTION]
[NO SQL | CONTAINS SQL | READS SQL DATA]
[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]
[INHERIT SPECIAL REGISTERS] [ [NOT] FEDERATED]
[ [NOT] FENCED [{THREADESAFE | NOT THREADESAFE}] ]
[ [NO] SCRATCHPAD inf] [ [NO] FINAL CALL]
    [DISALLOW PARALLEL]
[ [NO] DBINFO] [TRANSFORM GROUP group_name]
[CARDINALITY inf]
code_block
```

Following is an example external table UDF that retrieves message header information and some message text from Microsoft Exchange:

```
CREATE FUNCTION MAIL( )
RETURNS TABLE (time_rcvd DATE, subjct VARCHAR(15),
    lngth INTEGER, txt VARCHAR(30))
EXTERNAL NAME 'mail.header!list'
LANGUAGE OLE
PARAMETER STYLE SQL
NOT DETERMINISTIC FENCED CALLED ON NULL INPUT
```

SCRATCHPAD FINAL CALL

NO SQL

EXTERNAL ACTION DISALLOW PARALLEL

Note that the external function, **mail.header!list**, should already exist.

DB2 OLE-DB external table user-defined functions

This variation of the *CREATE FUNCTION* statement registers a user-defined OLE-DB external table function to access data from an OLE-DB provider. The table function may then be used in the *FROM* clause of a *SELECT* statement. This is useful to link to non-DB2 sources of relational data. An OLE-DB external table UDF allows the following clauses:

```
CREATE FUNCTION object_name (parameter_name datatype) [,...]
```

```
RETURNS TABLE datatype [,...] ] [SPECIFIC specific_name]
```

```
[LANGUAGE OLEDB] [EXTERNAL NAME external_routine_name]
```

```
[DETERMINISTIC | NOT DETERMINISTIC] [ [NO] EXTERNAL ACTION]
```

```
[STATIC DISPATCH] [CALLED ON NULL INPUT |
```

```
RETURNS NULL ON NULL INPUT]
```

```
[CARDINALITY inf]
```

```
code_block
```

The following example registers an OLE-DB table function that retrieves **order** information from the Microsoft Access database called **Northwind**:

```
CREATE FUNCTION orders ( )
```

```
RETURNS TABLE (orderid INTEGER, customerid CHAR(5), employeeid INTEGER,
```

```
orderdate TIMESTAMP, requireddate TIMESTAMP, shippeddate TIMESTAMP,
```

```
shipvia INTEGER, freight DEC(19,4))
```

```
LANGUAGE OLEDB
```

```
EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;
```

```
Data Source=c:\sql\lib\samples\oledb\nwind.mdb';
```

Note that the connection string is defined in the external name of the function.

DB2 sourced user-defined functions

This variation of the *CREATE FUNCTION* statement registers a user-defined function that is based on another existing scalar or column function. This statement can also be used to register a function template with an application server that acts as a federated server. (A *function template* is a partial function that has no executable code, but instead uses it to map to a data source function. Once the mapping is created, the user specifies the function template in queries to the federated server. The federated server will invoke the data source function and return values that correspond to those in the *RETURNS* portion of the template definition.) A source/template UDF allows the following clauses:

```
CREATE FUNCTION object_name (parameter_name datatype [,...])
```

```
RETURNS datatype [,...] ]
```

```
[SOURCE {object_name [input_argument [,...] ] |
```

```
[SPECIFIC specific_name] ]
```



```
[AS TEMPLATE [DETERMINISTIC | NOT DETERMINISTIC]
 [ [NO] EXTERNAL ACTION] ]
```

code_block

For example, the following sourced UDF is based on the **report_reader.center** scalar function created earlier in this section. The following function, called **fn_center**, accepts only integer arguments:

```
CREATE FUNCTION fn_center (INTEGER, INTEGER)
RETURNS FLOAT
SOURCE report_reader.center (INTEGER, FLOAT);
```

In the next example, using a federated server, we can invoke an Oracle UDF, called *statistics* in the schema *scott*, that returns table statistics as a double-precision floating point value:

```
CREATE FUNCTION scott.statistics (DOUBLE, DOUBLE)
RETURNS DOUBLE AS TEMPLATE
DETERMINISTIC NO EXTERNAL ACTION;
```

DB2 SQL user-defined functions

This variation of the *CREATE FUNCTION* statement defines a SQL scalar, table, or row function. This application shows the most common form of SQL UDF's in DB2. (Remember a *scalar function* returns a single value when invoked. A *table function* may be used in the *FROM* clause of a *SELECT* statement and returns a table. A *row function* returns a row and may be used as a transform function.)

A SQL UDF allows the following clauses:

```
CREATE FUNCTION object_name [(parameter_name datatype [,...])]
RETURNS {datatype | {ROW | TABLE} column_list}
    [SPECIFIC specific_name]
    [LANGUAGE SQL] [DETERMINISTIC | NOT DETERMINISTIC]
    [ [NO] EXTERNAL ACTION]
    [CONTAINS SQL | READS SQL DATA] [STATIC DISPATCH]
    [CALLED ON NULL INPUT] [PREDICATES (predicate_definition) ]
```

code_block

Here's an example of a SQL user-defined function that returns a concatenated name from separate fields:

```
CREATE FUNCTION prod.cust_name
(in_custno INTEGER )
RETURNS VARCHAR(31)
LANGUAGE SQL
READS SQL DATA
NO EXTERNAL ACTION
DETERMINISTIC
```

```
RETURN

SELECT (c.firstnme CONCAT ' '
        CONCAT c.midinit CONCAT ' '
        CONCAT c.lastname)
AS name
FROM prod.customer c
WHERE c.custno = 10;
```

DB2 external stored procedure

This variation of the *CREATE PROCEDURE* statement defines an external procedure with a DB2 application server. Usually, external procedures are written in a non-SQL language like C or Java. An external procedure allows the following clauses:

```
CREATE PROCEDURE object_name
  [( [IN | OUT | INOUT] parameter_name datatype [,...] ) ]
  [SPECIFIC specific_name] [DYNAMIC RESULT SETS int]
  [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
  [DETERMINISTIC | NOT DETERMINISTIC] [CALLED ON NULL INPUT]
LANGUAGE {C | JAVA | COBOL | OLE}
  EXTERNAL NAME external_routine_name
  [INHERIT SPECIAL REGISTERS]
PARAMETER STYLE {SQL | DB2GENERAL | DB2SQL | GENERAL |
  GENERAL WITH NULLS | JAVA}
  [ [NOT] FENCED [{THREADSAFE | NOT THREADSAFE}] ]
  [PROGRAM TYPE {SUB | MAIN}]
  [ [NO] DBINFO]
code_block
```

Following is an example of an external procedure. In the example, we define a procedure definition in DB2 for a C program that, when passed an author ID, tells us the total books sold, the total sales in dollars, and the book titles the specified author wrote:

```
CREATE PROCEDURE author_sales
  (IN au_id INTEGER, OUT books_sold INTEGER,
  OUT total_sales DOUBLE, OUT title VARCHAR(50) )
EXTERNAL NAME 'authors!sales'
DYNAMIC RESULT SETS 1 NOT FENCED
LANGUAGE C PARAMETER STYLE GENERAL
```

DB2 SQL stored procedure

This variation of the *CREATE PROCEDURE* statement defines a SQL-based procedure with a DB2 application server. A SQL procedure allows the following clauses:

```
CREATE PROCEDURE object_name [( [IN |  
    OUT| INOUT] parameter_name datatype [,...] ) ]  
    [SPECIFIC specific_name] [DYNAMIC RESULT SETS int]  
    [CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]  
    [DETERMINISTIC | NOT DETERMINISTIC] [CALLED ON NULL INPUT]  
    [INHERIT SPECIAL REGISTERS] [LANGUAGE SQL]  
code_block
```

The following SQL stored procedure returns a customer's record based upon a *SOUNDEX* function on their name and state:

```
CREATE PROCEDURE prod.custlookup  
    (IN in_lastname VARCHAR(15),  
    IN in_st CHARACTER(2) )  
    DYNAMIC RESULT SETS 1  
    LANGUAGE SQL  
    NOT DETERMINISTIC  
    CALLED ON NULL INPUT  
    MODIFIES SQL DATA  
    INHERIT SPECIAL REGISTERS  
P1: BEGIN  
    DECLARE cursor1 CURSOR WITH RETURN FOR  
        SELECT *  
        FROM prod.customer c  
        WHERE SOUNDEX (c.lastname) = SOUNDEX (in_lastname)  
        AND c.st = UPPER (in_st);  
    OPEN cursor1;  
END P1;
```

DB2 has recently implemented new procedural extensions to SQL. You can now define the *code_block* for a DB2 UDF or stored procedure using conditional processing constructs like *IF...THEN* and *WHILE* loops (as shown in the examples above).

RETURNS TABLE is available as a return value for any *LANGUAGE*. The only clauses allowed are *STATIC DISPATCH*, *DETERMINISTIC*, *EXTERNAL ACTION*, *RETURNS NULL ... | CALLED NULL... ON INPUT*, and *CARDINALITY*.

READ SQL DATA and *CONTAINS SQL* are the only SQL-style options allowed when you chose *LANGUAGE SQL*. These clauses disable the *CAST* clause and the *RETURNS NULL ON NULL INPUT* clause. It also makes available (though optional) the *RETURNS {ROW | TABLE}* clause, *FEDERATED* clause, *PREDICATES* clause, and *INHERIT SPECIAL REGISTERS* clause.

ALTER FUNCTION and *PROCEDURE* on DB2 merely change the characteristics of an existing UDF or stored procedure. The two examples shown next make a function and a procedure, respectively, *NOT FENCED*.

```
ALTER FUNCTION mail( ) NOT FENCED;
```

```
ALTER PROCEDURE employee_pay NOT FENCED;
```

MySQL

MySQL does not support the *ALTER FUNCTION* statement, nor does it support the *CREATE PROCEDURE* and *ALTER PROCEDURE* statements.

All user-defined functions created on a MySQL database are, in effect, external function calls:

```
CREATE [AGGREGATE] FUNCTION function_name
RETURNS {STRING | REAL | INTEGER}
SONAME shared_program_library_name ;
```

where:

```
CREATE [AGGREGATE] FUNCTION function_name
```

Creates an external function with a *function_name* of not more than 64 characters. The module will be stored in the *mysql2/udf* directory, with an extension of *.udf*. If *AGGREGATE* is specified, the user-defined function must return an aggregate value like *SUM()* or *AVG()*.

```
RETURNS {STRING | REAL | INTEGER}
```

Returns either *STRING* for character data, *REAL* for floating-point numbers, or *INTEGER* for whole numbers.

```
SONAME shared_program_library_name
```

Names the C/C++ program that performs the processing. This *shared_program* can be compiled and built into the MySQL engine, or it can function as an independent, linked program.

Once implemented, a MySQL UDF may be called just like any built-in function, such as *ABS()* or *SOUNDEX()*.

The implementation of *CREATE FUNCTION* in MySQL is dependent on procedural code in C/C++ under an operating system that supports dynamic loading. The C/C++ program is named in the *shared_program_library_name* option. The function may be compiled either directly into the MySQL server, making the function permanently available, or as a dynamically callable program. For example, the code behind the UDF created in the following statement might be found on a Unix server:

```
CREATE FUNCTION find_radius RETURNS INT SONAME "radius.so";
```

Oracle

Oracle supports *ALTER* and *CREATE* for both *FUNCTION* and *PROCEDURE* object types. (You may also wish to learn about Oracle packages, which can also be used to create UDFs and stored procedures. Check Oracle's documentation.) Oracle's *CREATE PROCEDURE* syntax is as follows:

```
CREATE [OR REPLACE] {FUNCTION | PROCEDURE} [schema.]object_name
[(parameter1 [IN | OUT | IN OUT] [NOCOPY] datatype [,...]) ] ]
RETURN datatype
[DETERMINISTIC] [AUTHID {CURRENT_USER | DEFINER}]
[PARALLEL_ENABLE [( PARTITION prtn_name BY {ANY | {HASH | RANGE}
(column [,...]) } {ORDER | CLUSTER} BY (column [,...]) ) ] ]
{ [PIPELINED | AGGREGATE] USING [schema.]implementation_type] |
[PIPELINED] {IS | AS} }
{code_block | LANGUAGE {JAVA NAME external_program_name |
```

```
C [NAME external_program_name]  
LIBRARY lib_name [AGENT IN (argument [,...])] [WITH CONTEXT]  
[PARAMETERS ( param [,...] ) ] ;
```

The *ALTER FUNCTION* and *ALTER PROCEDURE* statement shown next is used to recompile invalid UDFs or stored procedures:

```
ALTER {FUNCTION | PROCEDURE} [schema.]object_name  
COMPILE [DEBUG] [compiler_param = value [...]] [REUSE SETTINGS]
```

Following are the parameter descriptions:

CREATE [OR REPLACE] FUNCTION [*schema*.]*object_name*

Creates a new UDF or stored procedure. Use *OR REPLACE* to replace an existing procedure or UDF without first dropping it, and then having to reassign all permissions to it. Certain clauses are only used with user-defined functions, including the *RETURN* clause, the *DETERMINISTIC* clause, and the entirety of the *USING* clause.

[IN | OUT | IN OUT]

Specifies whether a parameter is an input to the function, an output from the function, or both.

NOCOPY

Speeds up performance when an *OUT* or *IN OUT* argument is very large, as with a *VARRAY* or *RECORD* datatype.

AUTHID {*CURRENT_USER* | *DEFINER*}

Forces the routine to run in the permission context of either the current user or the person who owns the function, using *AUTHID CURRENT_USER* or *AUTHID DEFINER*, respectively.

PARALLEL_ENABLE

Enables the routine to be executed by a parallel query operation on an SMP or parallel-processor server. This clause is used only for UDFs. (Don't use session state or package variables because they can't be expected to be shared among parallel execution servers.) Define the behavior of the *PARALLEL_ENABLE* query operation using these subclauses, where:

PARTITION *prtn_name* BY {ANY | {HASH | RANGE} (*column* [,...])}

Defines partitioning of inputs on functions with *REF CURSOR* arguments. This may benefit table functions. *ANY* allows random partitioning. You can restrict partitioning to a specific *RANGE* or *HASH* partition on a comma-delimited *column* list.

{ORDER | CLUSTER} BY (*column* [,...])

Orders or clusters parallel processing based on a comma-delimited *column* list. *ORDER BY* causes the rows to be locally ordered on the parallel execution server according to the *column* list. *CLUSTER BY* restricts the rows on the parallel execution server to the key values identified in the *column* list.

{PIPELINED | AGGREGATE} [USING [*schema*.] *implementation_type*]

PIPELINED iteratively returns the results of a table function, instead of the normal serial return of the *VARRAY* or nested table result set. This clause is only used for UDFs. The clause *PIPELINED USING implementation_type* is for an external UDF that uses a language like C++ or Java. The *AGGEGATE USING implementation_type* clause defines a UDF as an aggregate function (a function that evaluates many rows, but returns a single value).

IS | AS

Oracle treats *IS* and *AS* equally. Use either one to introduce the *code_block*.

code_block

Oracle allows a PL/SQL code block for user-defined functions and stored procedures. Alternately, you may use the *LANGUAGE* clause for stored procedures written in Java or C.

```
LANGUAGE {JAVA NAME external_program_name | C [NAME external_program_name] LIBRARY lib_name [AGENT IN  
(argument)] [WITH CONTEXT] [PARAMETERS ( params ) ] }
```

Defines the Java or C implementation of the external program. The parameters and semantics of each declaration are specific to Java and C, not SQL.

```
ALTER {FUNCTION | PROCEDURE} [schema.]object_name
```

Recompiles an invalid standalone stored routine. Use *CREATE ... OR REPLACE* to change the arguments, declarations, or definition of an existing routine.

```
COMPILE [DEBUG] [REUSE SETTINGS]
```

Recompiles the routine. Note that *COMPILE* is required. (You can see compile errors with the SQL*Plus command *SHOW ERRORS*.) The routine is marked valid if no compiler errors are encountered. The following optional subclauses may also be included with the *COMPILE* clause, where:

DEBUG

Generates and stores code used by the PL/SQL debugger.

```
compiler_param = value [...]
```

Specifies a PL/SQL compiler parameter. Allowable parameters include *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. Refer to Oracle's documentation on the PL/SQL compiler for more details.

REUSE SETTINGS

Maintains the existing compiler switch settings and reuses them for recompilation. Normally, Oracle drops and reacquires compiler switch settings.

In Oracle, UDFs and stored procedures are very similar in composition and structure. The primary difference is that a stored procedure cannot return a value to the invoking process, while a function may return a single value to the invoking process.

For example, you can pass in the name of a construction project to the following function to obtain the project's profit:

```
CREATE OR REPLACE FUNCTION project_revenue (project IN varchar2)
```

```
RETURN NUMBER
```

```
AS
```

```
    proj_rev NUMBER(10,2);
```

```
BEGIN
```

```
    SELECT SUM(DECODE(action,'COMPLETED',amount,0)) -
```

```
           SUM(DECODE(action,'STARTED',amount,0)) +
```

```
           SUM(DECODE(action,'PAYMENT',amount,0))
```

```
INTO proj_rev  
  
FROM construction_actions  
  
WHERE project_name = project;  
  
RETURN (proj_rev);  
  
END;
```

In this example, the user-defined function accepts the project name as an argument. Then it processes the project revenue, behind the scenes, by subtracting the starting costs from the completion payment and adding any other payments into the amount. The `RETURN (proj_rev);` line returns the amount to the invoking process.

In Oracle, UDFs *cannot* be used in the following situations:

- In a *CHECK* constraint or *DEFAULT* constraint of a *CREATE TABLE* or *ALTER TABLE* statement.
- In a *SELECT*, *INSERT*, *UPDATE*, or *DELETE*, the UDF cannot, either directly or indirectly (if invoked by another routine):
 - Have an *OUT* or *IN OUT* parameter. (Indirect calls may take *OUT* and *IN OUT* parameters.)
 - Terminate the transaction with *COMMIT*, *ROLLBACK*, *SAVEPOINT*, or a *CREATE*, *ALTER*, or *DROP* statement that implicitly issues a *COMMIT* or *ROLLBACK*.
 - Use session control (*SET ROLE*) or system control (the Oracle-specific statement *ALTER SYSTEM*) statements.
 - Write to a database (when a component of a *SELECT* statement or a parallelized *INSERT*, *UPDATE*, or *DELETE* statement).
 - Write to the same table that is modified by the statement that calls the UDF.

Note that when you recompile a routine with the *ALTER* statement, it is marked valid if no compiler errors are encountered. (It is marked invalid if any errors are encountered.) Perhaps more importantly, any objects that depend upon the recompiled routine are marked invalid, regardless of whether an error occurs. You can either recompile those dependent objects yourself, or you can allow Oracle to take some additional time to recompile them at runtime.

By way of example, the following recompiles the **project_revenue** function and maintains any compiler information for the PL/SQL debugger:

```
ALTER FUNCTION project_revenue COMPILE DEBUG;
```

PostgreSQL

PostgreSQL supports the *CREATE FUNCTION* statement, but not *CREATE PROCEDURE*, *ALTER FUNCTION*, or *ALTER PROCEDURE*. Since PostgreSQL does not have an internal procedural language, user-defined functions are usually external functions, though they may be simple SQL queries. The syntax to use to create a function is:

```
CREATE [OR REPLACE] FUNCTION function_name  
  
    ( [ parameter [, ...] ] )  
  
RETURNS datatype  
  
AS { code_block | object_file, link_symbol }  
  
LANGUAGE { C | SQL | PLPGSQL | PLTCL | PLTCLU | PLPERL |  
  
    internal }  
  
[WITH { [ISCACHEABLE] [,] [ISSTRICT] } ]
```

where:

CREATE [OR REPLACE] *function_name*

Creates a new function of the name you provide, or replaces an existing function. *OR REPLACE* does not enable you to change the name, input parameters, or output results of an existing function. You must drop and recreate a function to change any of those settings.

RETURNS *datatype*

Specifies the type of data required by the function.

AS {*code_block* | *object_file*, *link_symbol*}

Defines the composition of the user-defined function. The *code_block* can be a string defining the function (dependent on the *LANGUAGE* setting), such as an internal function name, the path and name of an object file, a SQL query, or the text of a procedural language. The definition also can be an *object file* and *link symbol* for a C-language function.

LANGUAGE {*C* | *SQL* | *PLPGSQL* | *PLTCL* | *PLTCLU* | *PLPERL* | *internal*}

Defines a call to an external program or an internal SQL routine. Since all of the language options, except *SQL*, are programs compiled in other languages, they are beyond the scope of this book. However, the *LANGUAGE SQL* clause should be used when writing *SQL* user-defined functions.



Add a new language not installed by default on PostgreSQL by using the *CREATE LANGUAGE* statement.

[WITH {[*ISCACHABLE*] [,] [*ISSTRICT*]}]

Optimizes PostgreSQL performance by indicating that the function always returns the same value when provided with the same parameter values. *WITH ISCACHABLE* is similar to the ANSI setting *DETERMINISTIC*, except that this setting then allows the optimizer to pre-evaluate the call of the function. *WITH ISSTRICT* is similar to the ANSI setting *RETURNS NULL ON NULL INPUT*. When omitted, the default behavior is similar to *CALLED ON NULL INPUT*. Note that you may include both keywords in a single function declaration.

PostgreSQL also allows function *overloading* in which the same function name is allowed for different functions, as long as they accept distinct input parameters.



Whenever you drop an existing function, all dependent objects are invalidated and must be recompiled.

Here's an example of a simple SQL function in PostgreSQL:

```
CREATE FUNCTION max_project_nbr
RETURNS int4
AS "SELECT MAX(title_ID) FROM titles AS RESULT"
LANGUAGE 'sql';
```

In this example, we created a UDF that returns the maximum **title_ID** from the **titles** table.

PostgreSQL uses *CREATE FUNCTION* as a substitute for *CREATE PROCEDURE*, as well as to define actions for *CREATE TRIGGER*.

SQL Server

SQL Server supports *CREATE* and *ALTER* for both types of routines. By default, SQL Server stored procedures can return result sets and SQL Server UDFs can return single or multirow result sets using the *TABLE* datatype on *RETURNS* arguments, contrary to the ANSI standard. However, this makes SQL Server routines more flexible and powerful. Use the following syntax to create a user-defined function or stored procedure:

```
CREATE {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]  
( [ {@parameter datatype [VARYING] [= default] [OUTPUT]} [,... ] )  
RETURNS {datatype | TABLE}  
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE |  
    RECOMPILE, ENCRPTION}]  
[FOR REPLICATION]  
[AS]  
    code_block
```

Use the following syntax to alter an existing user-defined function or stored procedure:

```
ALTER {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]  
( [ {@parameter datatype [VARYING] [= default] [OUTPUT]} [,... ] )  
RETURNS {datatype | TABLE}  
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE |  
    RECOMPILE, ENCRPTION}]  
[FOR REPLICATION]  
[AS]  
    code_block
```

Following are the parameter descriptions:

```
CREATE {FUNCTION | PROCEDURE} [owner_name.] object_name[;int]
```

Creates a new UDF or stored procedure in the current database. For SQL Server stored procedures, you may optionally specify a version number in the format *procedure_name;1*, where 1 is an integer indicating the version number. This facility allows you to have multiple versions of a single stored procedure.

```
{@parameter datatype [VARYING] [= default] [OUTPUT]} [,...]
```

Defines one or more input arguments for a UDF or stored procedure. SQL Server parameters are always declared with an at sign (@) as the first character.

where:

VARYING

Used in stored procedures with a *CURSOR* datatype parameter. Indicates that the result set is constructed dynamically by the procedure.

= *default*

Assigns a default value to the parameter. The default value is used whenever the stored procedure or UDF is invoked without supplying a value for the parameter.

OUTPUT

Used for stored procedures, *OUTPUT* is functionally equivalent to the ANSI *OUT* clause in the *CREATE FUNCTION* statement. The value stored in the return parameter is passed back to the calling procedure through the return variables of the SQL Server *EXEC[UTE]* command. Output parameters can be any datatype except *TEXT* and *IMAGE*.

RETURNS { *datatype* | TABLE }

Allows SQL Server UDFs to return a single *datatype* value or to return multiple values via the *TABLE* datatype. The *TABLE* datatype is considered *inline* if it has no accompanying column list and is defined with a single *SELECT* statement. If the *RETURN* clause returns multiple values via the *TABLE* datatype, and if the *TABLE* has defined columns and datatypes, the function is considered a *multistatement, table-valued* function.

WITH

Allows the assignment of additional characteristics to a SQL Server UDF or stored procedure.

ENCRYPTION

Tells SQL Server to encrypt the column in the system table that stores the text of the function, thus preventing unwarranted review of the function code. Usable by both UDFs and stored procedures.

SCHEMABINDING

Specifies that the function is bound to a specific database object, such as a table or view. That database object cannot be altered or dropped as long as the function exists (or maintains the *SCHEMABINDING* option). Usable only by UDFs.

RECOMPILE

Tells SQL Server not to store a cache plan for the stored procedure, but instead to recompile the cache plan each time the stored procedure is executed. This is useful when using atypical or temporary values in the procedure, but can cause serious degradation in performance. Used only by stored procedures. Note that *RECOMPILE* and *ENCRYPTION* can be invoked together.

FOR REPLICATION

Disables execution of the stored procedure on a subscribing server. It is used primarily to create a filtering stored procedure that is executed only by SQL Server's built-in replication engine. It is incompatible with *RECOMPILE*.

Like tables (see *CREATE TABLE*), local and global temporary stored procedures may be declared by prefixing a pound symbol (#) and double-pound symbol (##) to the name of the procedure, respectively. Temporary procedures exist only for the duration of the user or process session that created them. When that session ends, the temporary procedure automatically deletes itself.

A SQL Server stored procedure or UDF may have as many as 1,024 input parameters, specified by the at sign (@). Parameters are defined using SQL Server datatypes. (Parameters of *CURSOR* datatype must be both *VARYING* and *OUTPUT*.) The user or calling process must supply values for any input parameters. However, a default value can be supplied for any input parameter to allow the procedure to execute without a user- or process-supplied value. The default must be a constant or NULL, but it may contain wildcard characters, as discussed later in this chapter in the section [LIKE Operator](#).

SQL Server requires that one or more user-supplied parameters be declared for a given user-defined function. All SQL Server datatypes are supported as parameters, except *TIMESTAMP*. Values returned by the function can be any datatype except *TIMESTAMP*, *TEXT*, *NTEXT*, or *IMAGE*. If an inline table value is required, the *TABLE* option without an accompanying column list may be used.



The *ALTER FUNCTION* and *ALTER PROCEDURE* statements support the full syntax provided by the corresponding *CREATE* statements. You can use the *ALTER* statements to change any of the attributes of an existing routine without changing permissions or affecting any dependent objects.

For UDFs, the *code_block* is either a single *SELECT* statement for an inline function, in the format *RETURN (SELECT...)*, or a series of Transact-SQL statements following the *AS* clause for a multistatement operation. When using *RETURN (SELECT)*, the *AS* clause is optional. Some other rules for SQL Server UDFs:

- When using the *AS* clause, the *code_body* should be encapsulated in *BEGIN...END* delimiters.
- UDFs cannot make any permanent changes to data or cause other lasting side effects. A number of other restrictions exist as a result. For example, *INSERT*, *UPDATE*, and *DELETE* statements may modify only *TABLE* variables local to a function.
- When returning a scalar value, a SQL Server UDF must contain the clause *RETURN datatype*, where *datatype* is the same as that identified in the *RETURNS* clause.
- The last statement of the *code_block* must be an unconditional *RETURN* that returns a single datatype value or *TABLE* value.
- The *code_block* may not contain any global variables that return a perpetually changing value, such as *@@CONNECTIONS* or *GETDATE*. But it may contain those that return a single, unchanging value, such as *@@SERVERNAME*.

The following is an example of a *scalar* function that returns a single value. Once created, the scalar UDF can then be utilized in a query just like a system-supplied function. For example:

```
CREATE FUNCTION metric_volume -- Input dimensions in centimeters.
```

```
    (@length decimal(4,1),
```

```
    @width decimal(4,1),
```

```
    @height decimal(4,1) )
```

```
RETURNS decimal(12,3) -- Cubic Centimeters.
```

```
AS BEGIN
```

```
    RETURN ( @length * @width * @height )
```

```
END
```

```
GO
```

```
SELECT project_name,
```

```
    metric_volume(construction_height,
```

```
    construction_length,
```

```
    construction_width)
```

```
FROM housing_construction
```

```
WHERE metric_volume(construction_height,
```

```
    construction_length,
```

```
    construction_width) >= 300000
```

```
GO
```

An inline table-valued UDF supplies values via a single *SELECT* statement using an *AS RETURN* clause. For example, we can supply a store ID and find all of their **titles**:

```
CREATE FUNCTION stores_titles (@stor_id varchar(30))  
  
RETURNS TABLE  
  
AS  
  
RETURN (SELECT title, qty  
  
        FROM sales AS s  
  
        JOIN titles AS t ON t.title_id = s.title_id  
  
        WHERE s.stor_id = @storeid )
```

Now, let's alter the UDF just a bit by changing the input argument datatype length and adding another condition to the *WHERE* clause (indicated in boldface):

```
ALTER FUNCTION stores_titles  
  
(@stor_id VARCHAR(4))  
  
RETURNS TABLE  
  
AS  
  
RETURN (SELECT title, qty  
  
        FROM sales AS s  
  
        JOIN titles AS t ON t.title_id = s.title_id  
  
        WHERE s.stor_id = @storeid  
  
AND s.city = 'New York')
```

User-defined functions that return *TABLE* values are often selected as result set values or are used in the *FROM* clause of a *SELECT* statement, just as a regular table is used. These *multi-statement, table-valued functions* can have very elaborate code bodies since the *code_block* is composed of many Transact-SQL statements that populate a *TABLE* return variable.

Here is an example invoking a multistatement, table-valued function in a *FROM* clause. Notice that a table alias is assigned, just as for a regular table:

```
SELECT co.order_id, co.order_price  
  
FROM   construction_orders AS co,  
  
       fn_construction_projects('Cancelled') AS fcp  
  
WHERE  co.construction_id = fcp.construction_id  
  
ORDER BY co.order_id  
  
GO
```

For stored procedures, the *code_block* clause contains one or more Transact-SQL commands, up to a maximum size of 128 MB, delimited by *BEGIN* and *END* clauses. Some rules about Microsoft SQL Server stored procedures include:

- The *code_block* allows most valid Transact-SQL statements, but *SET SHOWPLAN_TEXT* and *SET SHOWPLAN_ALL* are prohibited.
- Some other commands have restricted usages within stored procedures, including *ALTER TABLE*, *CREATE INDEX*, *CREATE TABLE*, all *DBCC* statements, *DROP TABLE*, *DROP INDEX*, *TRUNCATE TABLE*, and *UPDATE STATISTICS*.

- SQL Server allows deferred name resolution, meaning that a stored procedure compiles without an error even though it references an object that has not yet been created. SQL Server creates the execution plan and fails only when the object is actually invoked, for instance in a stored procedure, if the object still doesn't exist.
- Stored procedures can be nested easily in SQL Server. Whenever a stored procedure invokes another stored procedure, the system variable @@NESTLEVEL is incremented by 1. It is decreased by 1 when the called procedure completes. Use *SELECT @@NESTLEVEL* inside a procedure or from an ad hoc query session to find the current nesting depth.

In the following example, a SQL Server stored procedure generates a unique 22-digit value (based on elements of the system date and time) and returns it to the calling process:

-- A Microsoft SQL Server stored procedure

```
CREATE PROCEDURE get_next_nbr
    @next_nbr CHAR(22) OUTPUT
AS
BEGIN
    DECLARE @random_nbr INT
    SELECT @random_nbr = RAND( ) * 1000000
    SELECT @next_nbr =
        RIGHT('000000' + CAST(ROUND(RAND(@random_nbr)*1000000,0))
            AS CHAR(6), 6) +
        RIGHT('0000' + CAST(DATEPART (yy, GETDATE( ) )
            AS CHAR(4)), 2) +
        RIGHT('000' + CAST(DATEPART (dy, GETDATE( ) )
            AS CHAR(3)), 3) +
        RIGHT('00' + CAST(DATEPART (hh, GETDATE( ) )
            AS CHAR(2)), 2) +
        RIGHT('00' + CAST(DATEPART (mi, GETDATE( ) )
            AS CHAR(2)), 2) +
        RIGHT('00' + CAST(DATEPART (ss, GETDATE( ) )
            AS CHAR(2)), 2) +
        RIGHT('000' + CAST(DATEPART (ms, GETDATE( ) )
            AS CHAR(3)), 3)
    END
GO
```

See Also

CALL
RETURN

CREATE/ALTER INDEX Statement

Indexes are special objects built on top of tables that speed many data-manipulation operations, such as *SELECT*, *UPDATE*, and *DELETE* statements. The selectivity of a given *WHERE* clause and the available query plans the database query optimizer can choose from is usually based upon the quality of indexes that have been placed on the table in a given database.

The *CREATE INDEX* command is not a part of the ANSI SQL standard, and thus varies greatly among vendors.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

Common Vendor Syntax

```
CREATE [UNIQUE] INDEX index_name ON table_name  
(column_name [, ...])
```

Keywords

```
CREATE [UNIQUE] INDEX index_name
```

Creates a new index named *index_name* in the current database and schema context. Since indexes are associated with specific tables (or sometimes views), the *index_name* need only be unique to the table it is dependent on. The *UNIQUE* keyword defines the index as a unique constraint for the table and disallows any duplicate values into the indexed column or columns of the table. (Refer to [Section 2.4.](#))

```
ON table_name
```

Declares the pre-existing table that the index is associated with. The index is dependent upon the table. If the table is dropped, so is the index.

```
(column_name [, ...])
```

Defines one or more columns in the table that are indexed. The pointers derived from the indexed column or columns enable the database query optimizer to greatly speed up data manipulation operations like *SELECT* and *DELETE* statements. All major vendors support *composite indexes*, also known as *concatenated indexes*. These indexes are used when two or more columns are best searched as a unit—for example, last name and first name.

Rules at a Glance

Indexes are created on tables to speed data manipulation operations that go against the tables such as those in a *WHERE* or *JOIN* clause. Indexes may also speed other operations, like:

- Identifying a *MIN()* or *MAX()* value in an indexed column.
- Sorting or grouping columns of a table.
- Searching based on *IS NULL* or *IS NOT NULL*.
- Fetching data much more quickly when the indexed data is all that is requested. A *SELECT* statement that retrieves data from an index and not directly from the table itself is called a *covering query*. An index that answers a query in this way is a *covering index*.

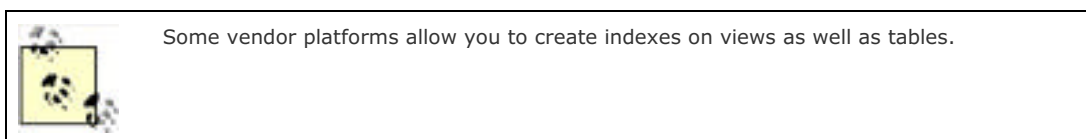
After creating a table, you can create indexes on columns within the table. It is a good idea to create indexes on columns that are frequently part of the *WHERE* clause or *JOIN* clause of the queries that go against a table. For example, the following statement creates an index on a column in the **sales** table that is frequently used in the *WHERE* clause of queries against that table:

```
CREATE INDEX ndx_ord_date ON sales(ord_date);
```

In another case, we want to set up the **pub_name** and **country** as a unique index on the **publishers** table:

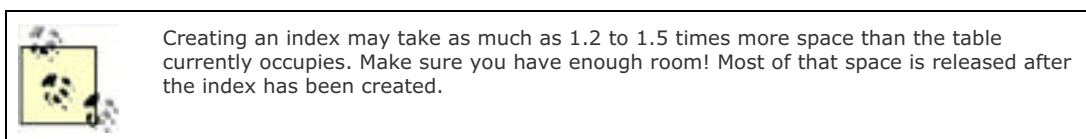
```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country);
```

Since the index is unique, any new record entered into the **publishers** table must have a unique combination of publisher name and country.



Programming Tips and Gotchas

Concatenated indexes are most useful when queries address the columns of the index starting from the left. If you omit left-side columns in a query against a concatenated key, the query may not perform as well. For example, assume that we have a concatenated index on (**last_name**, **first_name**). If we query only by **first_name**, the concatenated index that starts with **last_name** and includes **first_name** may not be any good to us. Some of the vendor platforms have advanced their query engines to the point where this is much less of a problem.



You should be aware that there are situations in which too many indexes can actually slow down system performance. In general, indexes greatly speed lookup operations, especially in *SELECT* statements, against a table or view. However, every index you create adds overhead whenever you perform an *UPDATE*, *DELETE*, or *INSERT* operation because the database must then update all dependent indexes with the values that have changed in the table. As a rule of thumb, six to twelve indexes are about the most you might want to create on a single table.

In addition, indexes take up extra space within the database. The more columns in an index, the more space it consumes. This is not usually a problem, but sometimes catches novices off guard when developing a new database.

Most databases use indexes to create statistical samplings, usually just called *statistics*, so that the query engine can quickly determine which, if any, index or combination of indexes will be most useful for a query. These indexes are always fresh and useful when the index is first created, but may become stale and less useful over time as records are deleted, updated, and inserted within the table. Consequently, indexes, like day-old bread, are not guaranteed to be useful as they age. You need to be sure to refresh, rebuild, and maintain your databases regularly to keep index statistics fresh.

DB2

DB2 supports the *CREATE INDEX* statement, but not the *ALTER INDEX* statement. (DB2 does have the *COMMENT ON INDEX* statement to allow users to update the comment for an index without dropping and recreating the index. Also, DB2 has a *RENAME INDEX* statement to rename an index with having to drop and recreate it.) The *CREATE INDEX* statement has additional clauses that allow partitioning based upon indexed columns, as well as other clauses that impact how data is physically arranged on disk:

```
CREATE [UNIQUE] INDEX index_name  
ON table_name ({column_name [ASC | DESC]} [, ...])
```

[SPECIFICATION ONLY]

[INCLUDE (*column_name* [ASC | DESC] [...])]

[{CLUSTER | EXTEND USING *index_extension* [(*constant* [...])] }]

[PCTFREE *int*] [MINPCTUSED *int*]

[[DIS]ALLOW REVERSE SCANS]

[COLLECT [SAMPLED [DETAILED]] STATISTICS]

where:

ASC | DESC

Specifies that the values in the index are kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default.

SPECIFICATION ONLY

Used to indicate that the value defined by *table_name* is a nickname; that is, it is the name of a table in a federated database. A DB2 federated database integrates data from heterogeneous data sources, such as tables on Oracle or SQL Server, and names them locally within the DB2 database as a nickname. To tell the DB2 optimizer about indexes on one of these nicknamed Oracle or SQL Server tables, create an index against the nickname and declare *SPECIFICATION ONLY*. DB2 then knows that the index exists and is available to the optimizer, but that it does not have to create the index.

INCLUDE (*column_name* [...])

Adds extra columns to a unique index for added selectivity, but any such included columns are not counted towards uniqueness. The main use for *INCLUDE* columns is to build *covering indexes*. Any columns added with the *INCLUDE* clause count towards the column and byte maximum allowed by DB2. *INCLUDE* is mutually exclusive of *EXTEND USING* and *SPECIFICATION ONLY* clauses, and cannot be used when indexing temporary tables.

CLUSTER

Declares that the index is clustered. A clustered index specifically controls the physical arrangement of the rows in the table to speed query I/O. The cluster factor is maintained and improved dynamically as data is inserted into the table. Only one clustered index is allowed per table. *CLUSTER* is mutually exclusive of *EXTEND USING* and *SPECIFICATION ONLY* clauses, and cannot be used with temporary tables or tables in append mode.

EXTEND USING *index_extension* [(*constant* [...])]

Names an index extension that manages the index. The index must be defined on a single column that is either a structured or a distinct type. One or more *constants* must be provided for any parameters required by the type. *EXTEND* cannot be used on a temporary table. Index extensions are very complex and somewhat uncommon. Refer to the vendor documentation for more detail.

PCTFREE *int*

Specifies a percentage (*int*) of space, which must be an integer of 0 to 99, to leave free on each index page of the index when it is created. When omitted, the default is 10. This option cannot be used on nicknames or temporary tables.

MINPCTUSED *int*

Defines the percentage (*int*) of free space, which must be an integer of 0 to 99, required on an index page before DB2 will consider merging index pages together. DB2 recommends a value of 50 or below so that the database will not spend excessive I/O merging index pages. This clause is usually effective only if background processes are able to gain an occasional table lock. If this is not possible due to heavy usage, consider using the DB2 command *REORG INDEXES*. This option cannot be used on nicknames or temporary tables.

[DIS]ALLOW REVERSE SCANS

Declares that the index can or cannot support scans under the order that the index was created in, with *ALLOW* and *DISALLOW*, respectively. The default is *DISALLOW*. This option cannot be used on nicknames.

COLLECT [SAMPLED [DETAILED]] STATISTICS

Collects index statistics during index creation. If the *SAMPLED* and *DETAILED* clauses are left off, then basic index statistics are created. The *DETAILED* clause collects extra statistics information for the index, namely *PAGE_FETCH_PAIRS* and *CLUSTERFACTOR*. The *SAMPLED* clause tells DB2 that sampling, a shortcut, can be used on extra statistics information from the *DETAILED* clause.

DB2 will not create a new index where an identical index exists. DB2 considers an index identical to another when one index contains the same columns in the same order as the other.

DB2 allows online use of a table while a new index is being created. Read access is always available during the index creation process, but write access is temporarily locked at the end of the index build process. Use the DB2 command *LOCK TABLE* to temporarily lock the table and avoid read access. You should also use the DB2 command *RUNSTATS* to update index statistics whenever a great deal of data has changed or after an index is created without the *COLLECT STATISTICS* clause included in the statement. The following two index creation examples are functionally the same:

-- collect basic index statistics at index creation

```
CREATE INDEX ndx_ord_date ON sales(ord_date) COLLECT STATISTICS;
```

-- collect index statistics after index creation

```
LOCK TABLE sales IN EXCLUSIVE MODE;
```

```
CREATE INDEX ndx_ord_date ON sales(ord_date);
```

```
RUNSTATS ON TABLE sales AND INDEX ndx_ord_date;
```

Now, we'll collect detailed statistics using sampling on the **publishers** table:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
```

```
COLLECT SAMPLED DETAILED STATISTICS;
```

You may wish to create an index on a nicknamed object. In the example below, the **publishers** nickname represents the **current_publishers** table:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
```

```
SPECIFICATION ONLY;
```

The following example shows how to create an extended indextype on a structured type column. The **coordinate_positions** UDT requires a literal value to maintain the **satellite_images** column:

```
CREATE INDEX satellite_images ON gis_sites(coordinates)
```

```
EXTEND USING (coordinate_positions ('x'00010010010000100020'));
```



DB2 also supports a statement called *CREATE INDEX EXTENSION*. It is used to enable user-defined indexing on typed tables. Refer to the vendor documentation for more details.

MySQL

MySQL supports a form of the *CREATE INDEX* statement, but not the *ALTER INDEX* statement. MySQL indexes are always stored in B-tree structures on the filesystem. Strings within an index are automatically prefix- and end-space-compressed. MySQL's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE | FULLTEXT] INDEX index_name  
ON table_name (column_name(length) [...])
```

where:

FULLTEXT

Creates a full-text search index against a column. Full-text indexes are only supported on MyISAM table types, *CHAR*, *VARCHAR*, or *TEXT* datatypes, and do not support the (*length*) clause.

MySQL supports the basic industry standard syntax for the *CREATE INDEX* statement. Interestingly, MySQL also lets you build an index on the first *length* characters of a *CHAR* or *VARCHAR* column. MySQL requires the (*length*) clause for *BLOB* and *TEXT* columns. Specifying a length can be useful when selectivity is sufficient in the first, say, 10 characters of a column, and in those situations where saving disk space is very important. For example:

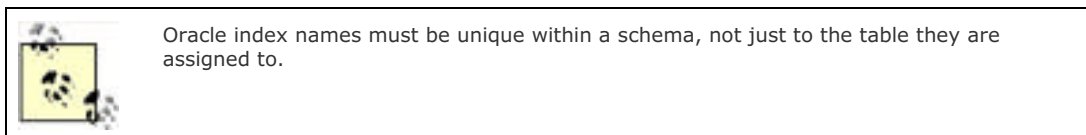
```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name(25),  
country(10))
```

This example indexes only the first 25 characters of the **pub_name** column and the first 10 characters of the **country** column.

As a general rule, MySQL allows at least 16 keys per table with a total maximum length of at least 256 bytes. This can vary by storage engine, however.

Oracle

Oracle allows the creation of indexes on tables, partitioned tables, clusters, and index-organized tables, as well as on scalar type object attributes of a typed table or cluster, and on nested table columns using the *CREATE INDEX* statement. Oracle also allows several types of indexes, including normal B-tree indexes, *BITMAP* indexes (useful for columns that have each value repeated 100 or more times), partitioned indexes, function-based indexes (based on an expression rather than a column value), and domain indexes.



Oracle also supports the *ALTER INDEX* statement. It is used to change or rebuild an existing index without forcing the user to drop and recreate the index. Oracle's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE | BITMAP] INDEX index_name  
{ON  
  {table_name ( {column | expression} [ASC | DESC] [...]) [{INDEXTYPE IS  
  index_type [PARALLEL [inf] | NOPARALLEL] [PARAMETERS ('values') ] } |  
  CLUSTER cluster_name |  
  FROM table_name WHERE condition [LOCAL partitioning] }  
[ {LOCAL partitioning | GLOBAL partitioning} ]  
[physical_attributes_clause] [{LOGGING | NOLOGGING}] [ONLINE]  
[COMPUTE STATISTICS] [{TABLESPACE tablespace_name | DEFAULT}]  
[{COMPRESS int | NOCOMPRESS}] [{NOSORT | SORT}] [REVERSE]  
[{PARALLEL [inf] | NOPARALLEL}]
```

The syntax for *ALTER INDEX* is:

```
ALTER INDEX index_name
{ {ENABLE | DISABLE} | UNUSABLE | RENAME TO new_index_name | COALESCE |
[NO]MONITORING USAGE | UPDATE BLOCK REFERENCES |
PARAMETERS ( 'ODCI_params' ) | alter_index_partitioning_clause |
rebuild_clause |
[DEALLOCATE UNUSED [KEEP int [K | M | G | T] ]
[ALLOCATE EXTENT ( [SIZE int [K | M | G | T] ] [DATAFILE 'filename'
[INSTANCE intf ] ]
[SHRINK SPACE [COMPACT] [CASCADE] ]
[{PARALLEL [intf] | NOPARALLEL}]
[{LOGGING | NOLOGGING}]
[physical_attributes_clause] }
```

where the non-ANSI clauses are:

BITMAP

Creates an index bitmap for each index value, rather than indexing each individual row. Bitmaps are best for low concurrency tables, like read-intensive tables. *BITMAP* indexes are incompatible with global partitioned indexes, the *INDEXTYPE* clause, or index-organized tables without a mapping table association.

ASC | DESC

Specifies that the values in the index are kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default. However, be aware that Oracle treats *DESC* indexes as function-based indexes, so there is some difference in functionality between *ASC* indexes and *DESC* indexes. You may not use *ASC* or *DESC* when also using the *INDEXTYPE* clause. *DESC* is ignored on *BITMAP* indexes.

INDEXTYPE IS index_type [PARAMETERS ('*values*')]]

Creates an index on a user-defined type of *index_type*. Domain indexes require that the user-defined type already exists (refer to the [CREATE/ALTER TYPE Statement](#) section). If the user-defined type requires arguments, pass them in using the optional *PARAMETERS* clause. You may also optionally parallelize the creation of the type index using the *PARALLEL* clause (explained below in more detail).

CLUSTER cluster_name

Declares a clustering index based on the specified pre-existing *cluster_name*. On Oracle, a clustering index physically co-locates two tables that are frequently queried on the same columns, usually a primary key and a foreign key. (Clusters are created with the Oracle-specific command *CREATE CLUSTER*.) You do not declare a table or columns on a *CLUSTER* index, since both the tables involved and the columns indexed are already declared with the previously issued *CREATE CLUSTER* statement.

GLOBAL partitioning

Includes the full syntax of:

GLOBAL PARTITION BY

```
{RANGE (column_list) ( PARTITION [partition_name] VALUE LESS THAN
(value_list) [physical_attributes_clause] [TABLESPACE
```

```
tablespace_name] [LOGGING | NOLOGGING] } [,...] ), |  
HASH (column_list) ( PARTITION [partition_name]  
{[TABLESPACE tablespace_name] [[OVERFLOW] TABLESPACE  
tablespace_name] [VARRAY varray_name STORE AS {LOB  
lob_segment_name] [LOB (lob_name) STORE AS [lob_segment_name]  
[TABLESPACE tablespace_name] } } |  
[STORE IN (tablespace_name [,...])] [OVERFLOW STORE IN  
(tablespace_name [,...]) , } [,...] }
```

The *GLOBAL PARTITION* clause declares that the global index is manually partitioned via either range or hash partitioning onto *partition_name*. (The default is to partition the index equally in the same way the underlying table is partitioned, if at all.) You can specify a maximum of 32 columns, though none may be ROWID. You may also apply the *[NO]LOGGING* clause, *TABLESPACE* clause, and the *physical_attributes_clause* (defined earlier) to a specific partition. You cannot partition on ROWID. You may include one or more partitions, along with any attributes, in a comma-delimited list, according to the following:

RANGE

Creates a range-partitioned global index based on the range of values from the table columns listed in the *column_list*.

VALUE LESS THAN (value_list)

Sets an upper bound for the current partition of the global index. The values in the *value_list* correspond to the columns in the *column_list*, both of which are comma-delimited lists of columns. Both lists are prefix-dependent, meaning that for a table with columns (*a, b, c*), you could define partitioning on (*a, b*) or (*a, b, c*), but not (*b, c*). The last value in the list should always be the keyword *MAXVALUE*.

HASH

Creates hash-partitioned global index assigned rows in the index to each partition based on a hash function of the values of the columns in the *column_list*. You may specify the exact tablespace to store special database objects such as *VARRAYs*, and *LOBs*, and for any *OVERFLOW* of the specified (or default) tablespaces.

LOCAL partitioning

Supports local index partitioning on *range*-partitioned indexes, *list*-partitioned indexes, *hash*-partitioned indexes, and *composite*-partitioned indexes. You may include zero or more partitions, along with any attributes, in a comma-delimited list. When omitted, Oracle generates one or more partitions consistent with those of the table partition. Index partitioning is done in one of three ways:

Range- and list-partitioned indexes

Applied to regular or equipartitioned tables. Range- and list-partitioned indexes (synonyms for the same thing) follow the syntax:

```
LOCAL [ (PARTITION [partition_name]  
{ [physical_attributes_clause] [TABLESPACE tablespace_name]  
[LOGGING | NOLOGGING] |  
[COMPRESS | NOCOMPRESS] } [,...] ) ]
```

All of the options are the same as for *GLOBAL PARTITION* (earlier), except that the scope is for a local index.

Hash-partitioned indexes

Applied to hash-partitioned tables. Hash-partitioned indexes allow you to choose between the earlier syntax and the optional syntax:

```
LOCAL {STORE IN (tablespace_name [...]) } |  
  
(PARTITION [partition_name [TABLESPACE tablespace_name] ] }
```

to store the index partition on a specific tablespace. When you supply more tablespace names than index partitions, Oracle will cycle through the tablespaces when it partitions the data.

Composite-partitioned indexes

Composite-partitioned indexes can be applied on composite-partitioned tables using the following syntax:

```
LOCAL [STORE IN (tablespace_name [...]) ]  
  
PARTITION [partition_name  
  
{ [physical_attributes_clause] [TABLESPACE tablespace_name]  
  
[LOGGING | NOLOGGING] |  
  
[COMPRESS | NOCOMPRESS] }  
  
[ {STORE IN (tablespace_name [...]) } |  
  
(SUBPARTITION [subpartition_name [TABLESPACE  
  
tablespace_name]) } ]
```

You may use the *LOCAL STORE* clause shown under the hash-partitioned indexes entry or from the *LOCAL* clause under the range- and list-partitioned indexes entry. (When using the *LOCAL* clause, substitute the keyword *SUBPARTITION* for *PARTITION*.)

physical_attributes_clause

Establishes values for one or more of the following settings: *PCTFREE int* (discussed here), *INITTRANS int*, and *STORAGE* (discussed in the [CREATE/ALTER TABLE Statement](#) section). When omitted, Oracle defaults to *PCTFREE* 10 and *INITTRANS* 2. The default for *MAXTRANS* depends on the system block size, according to the following:

PCTFREE int

Designates the percentage of free space to leave on each block of the index as it is created. This speeds up new entries and updates on the table. However, *PCTFREE* is only applied when the index is created. It is not maintained over time. Therefore, the amount of free space can erode over time as records are inserted, updated, and deleted from the index. This clause is not allowed on index-organized tables.

INITTRANS int

Designates the initial number of concurrent transactions allocated to each data block of the database. The value may range from 1 to 255. If a *physical_attributes_clause* is omitted, Oracle defaults to *PCTFREE* 10 and *INITTRANS* 2.

LOGGING | NOLOGGING

Tells Oracle to log the creation of the index on the redo logfile (*LOGGING*), or to not log it (*NOLOGGING*). This clause also sets the default behavior for subsequent bulk loads using Oracle SQL* Loader. For partitioned indexes, this clause establishes the default value of all partitions, segments associated with the partitions, and the defaults used on any partitions or subpartitions later added with an *ALTER TABLE...ADD PARTITION* statement. (When using *NOLOGGING*, we recommend that you take a full backup after the index has been loaded in case the index must be rebuilt due to a failure.)

ONLINE

Allows data manipulation on the table while the index is being created. Even when *ONLINE*, there is a very small window at the end of the create index operation where the table will be locked while the index creation completes. Any changes made to the base table at that time will then be reflected in the newly created index. *ONLINE* is incompatible with bitmap, cluster, or parallel clauses. It also cannot be used on indexes on a **UROWID** column or on index-organized tables with more than 32 columns in their primary keys.

COMPUTE [STATISTICS]

Collects statistics while the index is created, when it can be done at relatively little cost. Otherwise, you will have to collect statistics after the index is created.

TABLESPACE tablespace_name | DEFAULT

Assigns the index to a specific tablespace. When omitted, the index is placed on the default tablespace. Use the *DEFAULT* keyword to explicitly place an index into the default tablespace. (When local partitioned indexes are placed in *TABLESPACE DEFAULT*, the index partition (or subpartition) is placed in the corresponding tablespace of the base table partition (or subpartition).

COMPRESS [int] | NOCOMPRESS

Enables or disables key compression, respectively. Compression eliminates repeated occurrences of key column values, yielding a substantial space savings at the cost of speed. The integer value, *int*, defines the number of prefix keys to compress. The value can range from 1 to the number of columns in the index for nonunique indexes, and from 1 to *n*-1 columns for unique indexes. The default is *NOCOMPRESS*, but if you specify *COMPRESS* without an *int* value, the default is *COMPRESS n* (for nonunique indexes) or *COMPRESS n-1* (for unique indexes), where *n* is the number of columns in the index. *COMPRESS* can only be used on nonpartitioned and nonbitmapped indexes.

NOSORT | REVERSE

NOSORT an index to be created quickly for a column that is already sorted in ascending order. If the values of the column are not in perfect ascending order, the operation aborts, allowing a retry without the *NOSORT* option. *REVERSE*, by contrast, places the index blocks in storage by reverse order (excluding *rowid*). *REVERSE* is mutually exclusive of *NOSORT* and cannot be used on a bitmap index or an index-organized table. *NOSORT* is most useful for creating indexes immediately after a base table is loaded with data in pre-sorted order.

PARALLEL [int] | NOPARALLEL

Allows the parallel creation of the index using multiple server process, each operating on a distinct subset of the index, to speed up the operation. An optional integer value, *int*, may be supplied to define the exact number of parallel threads used in the operation. When omitted, Oracle calculates the number of parallel threads to use. *NOPARALLEL*, the default, causes the index to be created serially.

{ENABLE | DISABLE}

Enables or disables a pre-existing function-based index, respectively. You cannot specify any other clauses of the *ALTER INDEX* statement with *ENABLE* or *DISABLE*.

UNUSABLE

Marks the index (or index partition or subpartition) as unusable. When *UNUSABLE*, an index (or index partition or subpartition) may only be rebuilt or dropped and recreated before it can be used.

RENAME TO new_index_name

Renames the index from *index_name* to *new_index_name*.

COALESCE

Merges the contents of index blocks used to maintain the index-organized table so that the blocks can be reused. *COALESCE* is similar to *SHRINK*, though *COALESCE* compacts the segments less densely than does *SHRINK*, and it does not release unused space.

[NO]MONITORING USAGE

Declares that Oracle should clear existing information on index usage and monitor the index, posting information in the *V\$OBJECT_USAGE* dynamic performance view, until *ALTER INDEX ... NOMONITOR USAGE* is executed.

UPDATE BLOCK REFERENCES

Updates all stale guess data block addresses stored as part of the index row on normal or domain indexes of an index-organized table. The guess data blocks contain the correct database addresses for the corresponding blocks identified by the primary key. This clause cannot be used with other clauses of the *ALTER INDEX* statement.

PARAMETERS ('*ODCI_params*')

Specifies a parameter string passed, without interpretation, to the ODCI indextype routine of a domain index. The parameter string, called '*ODCI_params*', may be up to 1,000 characters long. Refer to the vendor documentation for more information on ODCI parameter strings.

alter_index_partitioning_clause

Refer to [Oracle partitioned and subpartitioned tables](#) in the [CREATE/ALTER TABLE Statement](#) section for more details.

rebuild_clause

Rebuilds the index or a specific partition (or subpartition) of the index. A successful rebuild marks an *UNUSABLE* index as *USABLE*. The syntax for the *rebuild_clause* is:

REBUILD {[*NO*]REVERSE | [*SUB*]PARTITION *partn_name*}

[{PARALLEL [*int*] | NOPARALLEL}] [TABLESPACE *tablespace_name*]

[PARAMETERS ('*ODCI_params*')] [ONLINE] [COMPUTE STATISTICS]

[COMPRESS *int* | NOCOMPRESS] [[*NO*]LOGGING]

[*physical_attributes_clause*]

where:

[NO]REVERSE

Stores the bytes of the index block in reverse order and excludes rows when the index is rebuilt (*REVERSE*) or stores the bytes of the index blocks in regular order (*NOREVERSE*).

DEALLOCATE UNUSED [KEEP *int* [K | M | G | T]]

Deallocates unused space at the end of the index (or at the end of each range- or hash-partition of a partitioned index) and frees the space for other segments in the tablespace. The optional *KEEP* keyword defines how many bytes, *int*, above the high-water mark the index will keep after deallocation. You can append a suffix to the *int* value to indicate that the value equals kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). When the *KEEP* clause is omitted, all unused space is freed.

ALLOCATE EXTENT ([SIZE *int* [K | M | G | T]] [DATAFILE '*filename*'] [INSTANCE *int*])

Explicitly allocates a new extent for the index using the specified parameters. You may mix and match any of the parameters. *SIZE* specifies the amount of next extent. The size of the extents may be in bytes (no suffix), kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). *DATAFILE* allocates an entirely new datafile to the index extent. *INSTANCE*, used only on Oracle Real Application Clusters, makes a new extent available to a freelist group associated with the specified instance.

SHRINK SPACE [COMPACT] [CASCADE]

Shrinks the index segments, though only segments in tablespaces with automatic segment management may be shrunk. Shrinking a segment moves rows in the table, so make sure *ENABLE ROW MOVEMENT* is also used in the *ALTER TABLE...SHRINK* statement. Oracle compacts the segment, releases the emptied space, and adjusts the high-water mark unless the optional keywords *COMPACT* and/or *CASCADE* are applied. The *COMPACT* keyword only defragments the segment space and compacts the index, but it does not readjust the high-water mark nor empty the space immediately. The *CASCADE* keyword performs the same shrinking operation (with some restrictions and exceptions) on all dependent objects of the index. The statement *ALTER INDEX ... SHRINK SPACE COMPACT* is functionally equivalent to *ALTER INDEX ... COALESCE*.

By default, Oracle indexes are non-unique. It is also important to know that Oracle's regular B-tree indexes do not include records that have a null key value.

Oracle does not support indexes on columns with the following datatypes: *LONG*, *LONG RAW*, *REF* (with the *SCOPE* attribute), or user-defined datatypes. You may create indexes on functions and expressions, but they cannot allow null values or aggregate functions. When you create an index on a function, the function, if it has no parameters, should show an empty set. For example: *function_name()*. If the function is a UDF, it must be *DETERMINISTIC*.

Oracle supports a special index structure called an *index-organized table* (IOT). IOTs combine the table data and primary key index on a single physical structure, instead of having separate structures for the table and for the index. IOTs are created using the *CREATE TABLE...ORGANIZATION INDEX* statement. Refer to [CREATE/ALTER TABLE Statement](#) for more information on making an IOT.

Oracle automatically creates any additional indexes on an index-organized table as secondary indexes. Secondary indexes do not support the *REVERSE* clause.

Oracle allows the creation of partitioned indexes and tables with the *PARTITION* clause. Consequently, Oracle's indexes also support partitioned tables. The *LOCAL* clause tells Oracle to create separate indexes for each partition of a table. The *GLOBAL* clause tells Oracle to create a common index for all the partitions.

Note that any time an object name is referenced in the syntax diagram, you may optionally supply the schema. This applies to indexes, tables, etc., but not to tablespaces. You must have the explicitly declared privilege to create an index in a schema besides the current one.

As an example, you can use a statement such as the following to create an Oracle index that is compressed, created in parallel, and with compiled statistics, but without logging the creation:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
COMPRESS 1 PARALLEL NOLOGGING COMPUTE STATISTICS;
```

As with other Oracle object creation statements, you can control how much space they consume and in what increments they grow. The following example constructs an index in Oracle on a specific tablespace with specific instructions for how the data is to be stored:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
STORAGE (INITIAL 10M NEXT 5M PCTINCREASE 0)
TABLESPACE publishers;
```

For example, when you create the **housing_construction** table as a partitioned table on an Oracle server, you should also create a partitioned index with its own index partitions:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
ON housing_construction(project_id)
GLOBAL PARTITION BY RANGE (project_id)
(PARTITION part1 VALUES LESS THAN ('H')
TABLESPACE construction_part1_ndx_ts,
PARTITION part2 VALUES LESS THAN ('P')
TABLESPACE construction_part2_ndx_ts,
PARTITION part3 VALUES LESS THAN (MAXVALUE)
TABLESPACE construction_part3_ndx_ts);
```


If in fact the **housing_construction** table used a composite partition, we could accommodate that here:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
ON housing_construction(project_id)
STORAGE (INITIAL 10M MAXEXTENTS UNLIMITED)
LOCAL (PARTITION part1 TABLESPACE construction_part1_ndx_ts,
PARTITION part2 TABLESPACE construction_part2_ndx_ts
(SUBPARTITION subpart10, SUBPARTITION subpart20,
SUBPARTITION subpart30, SUBPARTITION subpart40,
SUBPARTITION subpart50, SUBPARTITION subpart60),
PARTITION part3 TABLESPACE construction_part3_ndx_ts);
```

In the following example, we could rebuild the **project_id_ind** index that was created earlier by using parallel execution processes to scan the old index and build the new index in reverse order:

```
ALTER INDEX project_id_ind
REBUILD REVERSE PARALLEL;
```

Similarly, we could split out an additional partition on **project_id_ind**:

```
ALTER INDEX project_id_ind
SPLIT PARTITION part3 AT ('S')
INTO (PARTITION part3_a TABLESPACE constr_p3_a LOGGING,
PARTITION part3_b TABLESPACE constr_p3_b);
```

PostgreSQL

PostgreSQL allows the creation of ascending-order indexes, as well as *UNIQUE* indexes. Its implementation also includes a performance enhancement under the *USING* clause. PostgreSQL's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE] INDEX index_name ON table_name
[USING {BTREE | RTREE | HASH} ]
{function_name | (column_name [,...]) }
[WHERE condition]
```

where:

USING [BTREE | RTREE | HASH | GIST]

Specifies one of three dynamic access methods to optimize performance. Of particular importance is the fact that the indexes are fully dynamic and do not require periodic refreshes of their statistics. The *USING* options are as follows:

BTREE

Uses Lehman-Yao's high-concurrency B-tree structures to optimize the index. This is the default method when no other is specified. B-tree indexes, the default, can be invoked for comparisons using =, <, <=, >, and >=. B-tree supports multicolumn indexes.

RTREE

Uses Guttman's quadratic-split algorithm R-tree structures to optimize the index. R-tree indexes can be invoked for comparison using <<, &<, &>, >>, @, ~=, and &&. R-tree indexes must be single-column indexes.

HASH

Uses Litwin's linear hashing algorithm to optimize the index. Hash indexes can be invoked for comparisons using =. Hash indexes must be single-column indexes.

GIST

Uses Generalized Index Search Trees (GISTs) to optimize the index. GIST indexes may be multicolumn.

function_name

Defines a user-defined function, rather than a column from the base table, as the basis of the index values. A function-based index is mutually exclusive of a regular column-based index.

WHERE *condition*

Defines a *WHERE* clause search condition, which is then used to generate a *partial* index. A partial index contains entries for a select set of records in the table, not all records of the table. You can get some interesting effects from this clause. For example, you can pair *UNIQUE* and *WHERE* to enforce uniqueness for a subset of the table rather than for the whole table. The *WHERE* clause must:

- Reference columns in the base table (though they need not be columns of the index itself).
- Not make use of aggregate functions.
- Not use subqueries.

In PostgreSQL, a column may have an associated *operator class* based on the datatype of the column. An operator class specifies the operators for a particular index. Although users are free to define any valid operator class for a given column, the default operator class is the appropriate operator class for that column type.

In the following example, we create an index using the *GIST* indextype. We also make sure that uniqueness is only enforced for publishers outside of the USA:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
USING GIST
WHERE country <> 'USA';
```

SQL Server

SQL Server's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE] [[NON]CLUSTERED] INDEX index_name
ON {table_name | view_name} (column [ASC | DESC] [,...])
[WITH [PAD_INDEX]
    [,] FILLFACTOR = int]
    [,] IGNORE_DUP_KEY]
    [,] DROP_EXISTING]
    [,] STATISTICS_NORECOMPUTE ]
    [,] SORT_IN_TEMPDB ]
[ON filegroup]
```

where:

[NON]CLUSTERED

Controls the physical ordering of data for the table using either a *CLUSTERED* or *NONCLUSTERED* index. The columns of a *CLUSTERED* index determine the order in which the records of the table are physically written. Thus, if you create an ascending clustered index on column A of table **Foo**, the records will be written to disk in ascending alphabetical order. The *NONCLUSTERED* clause, the default, when either value is omitted, creates a secondary index containing only pointers and has no impact on how the actual rows of the table are written to disk.

ASC | DESC

Specifies that the values in the index are kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default.

WITH

Allows the specification of one or more optional attributes for the index.

PAD_INDEX

Specifies that space should be left open on each 8K index page, according to the value established by the *FILLFACTOR* setting.

FILLFACTOR = int

Declares a percentage value, *int*, from 1 to 100 that tells SQL Server how much of each 8K data page should be filled at the time the index is created. This is useful to reduce I/O contention and page splits when a data page fills up. Creating a clustered index with an explicitly defined fillfactor can increase the size of the index, but also speed up processing in certain circumstances.

IGNORE_DUP_KEY

Controls what happens when a duplicate record is placed into a unique index through an insert or update operation. If this value is set for a column, only the duplicate row is excluded from the operation. If this value is not set, then all records in the operation (even nonduplicate records) are rejected as duplicates.

DROP_EXISTING

Drops any pre-existing indexes on the table and rebuilds the specified index.

STATISTICS_NORECOMPUTE

Stops SQL Server from recomputing index statistics. This can speed the *CREATE INDEX* operation, but it may mean that the index is less effective.

SORT_IN_TEMPDB

Stores any intermediate results used to build the index in the system database, *TEMPDB*. This increases the space needed to create the index, but could speed processing if *TEMPDB* is on a different disk than the table and index.

ON filegroup

Creates the index on a given pre-existing filegroup. This enables the placing of indexes on a specific hard disk or RAID device. Issuing a *CREATE CLUSTERED INDEX...ON FILEGROUP* effectively moves a table to the new filegroup, since the leaf level of the clustered index is the same as the actual data pages of the table.

SQL Server allows the creation of a unique clustered index on a view, effectively materializing the view. This can greatly speed up data retrieval operations against the view. Once a view has a unique clustered index, nonclustered indexes also can be added to the view. Note that the view also must be created using the *SCHEMABINDING* option. Indexed views are only allowed in SQL Server 2000 Enterprise Edition, unless you add a *NOEXPAND* hint to the view. Indexed views support data retrieval, but not data modification. (See the [SQL Server](#) section of [CREATE/ALTER VIEW Statement](#) for specific restrictions on indexed views.)

SQL Server allows up to 249 nonclustered indexes (unique or non-unique) on a table, as well as one primary key index. Index columns may not be of datatype *NTEXT*, *TEXT*, or *IMAGE*. Concatenated keys may contain up to 16 columns and/or a total of 900 bytes across all fixed-length columns.

SQL Server automatically parallelizes the creation of the index according to the configuration option *max degree of parallelism*.

It is often necessary to build indexes that span several columns—i.e., a *concatenated key*. Here is an example:

```
CREATE UNIQUE INDEX project2_ind
ON housing_construction(project_name, project_date)
WITH PAD_INDEX, FILLFACTOR = 80
ON FILEGROUP housing_fg
GO
```

Adding the *PAD_INDEX* clause and setting the *FILLFACTOR* to 80 tells SQL Server to leave the index and data pages 80% full, rather than 100% full. The example also tells SQL Server to create the index on the **housing_fg** filegroup, rather than the default filegroup.

See Also

CREATE/ALTER TABLE
DROP

CREATE/ALTER METHOD Statement

The *CREATE/ALTER METHOD* statements allow the creation of a new database method or the alteration of an already existing database method. An easy (but loose) way to think of a method is that it is a user-defined function associated with a user-defined type. For example, a method called **Office**, of type **Address**, can accept a *VARCHAR* input parameter and passes out a result of **Address**.

An implicitly defined method is created every time a structured type is created (see [CREATE/ALTER TYPE Statement](#) for more details). Using a combination of the *CREATE TYPE* statement and the *CREATE METHOD* statement creates user-defined methods.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
{CREATE | ALTER} [INSTANT | STATIC] METHOD method_name
( [ [{IN | OUT | INOUT}] param datatype [AS LOCATOR] [RESULT] [...]] )
RETURNS datatype
```

FOR *udt_name*

[SPECIFIC *specific_name*]

code_body

Keywords

CREATE | ALTER [INSTANT | STATIC] *method_name*

Creates a new method or alters an existing method and, optionally, specifies it as an *INSTANT* or *STATIC* method.

[(IN | OUT | INOUT) *param datatype* [,...]

Declares one or more parameters to be passed into the method in a comma-delimited list enclosed in parentheses. Parameters used with a method may pass a value *IN*, *OUT*, or both in and out via *INOUT*. The syntax for the parameter declaration is:

[{IN | OUT | INOUT}] *parameter_name1 datatype*,

[{IN | OUT | INOUT}] *parameter_name2 datatype*,[...]

Make sure the name is unique within the method. When used with *ALTER*, this clause adds parameters to a pre-existing method. Refer to [Chapter 2](#) for details on datatypes.

AS *LOCATOR*

The optional *AS LOCATOR* clause is used to validate an external routine with a *RETURNS* parameter that is a *BLOB*, *CLOB*, *NCLOB*, *ARRAY*, or user-defined type. In other words, the locator (i.e., a pointer) for the *LOB* is returned but not the entire value of the *LOB*.

RESULT

Designates a user-defined type, but is not needed for standard datatypes.

RETURNS *datatype*

Declares the datatype of the results returned by a method. The key purpose of a user-defined method is to return a value. If you need to change the datatype of a *RETURNS* parameter on the fly, use the *CAST* clause (refer to the function *CAST*). For example, *RETURNS VARCHAR(12) CAST FROM DATE*.

FOR *udt_name*

Associates the method with a specific, pre-existing user-defined type, created using *CREATE TYPE*.

SPECIFIC *specific_name*

Uniquely identifies the function generally used with user-defined types.

Rules at a Glance

User-defined methods are essentially a different approach to the same output provided by user-defined functions. For example, consider the following two pieces of code:

```
CREATE FUNCTION my_fcn (order_udt)
RETURNS INT;
CREATE METHOD my_mthd ( )
RETURNS INT
FOR order_udt;
```

Although the code sections for the function and method are different, they do exactly the same thing.

The rules for use and invocation of methods are otherwise the same as for functions.

Programming Tips and Gotchas

The main difficulty with methods is that they are an object-oriented approach to the same sort of functionality provided by user-defined functions. Since they accomplish the same work, using a different approach, it can be hard to decide which approach to use.

DB2

DB2 supports both the *CREATE METHOD* and *ALTER METHOD* statements:

```
CREATE [SPECIFIC] METHOD method_name
( [ [{IN | OUT | INOUT}] [param] datatype [AS LOCATOR] [RESULT] [...]] )
[RETURNS datatype [{AS LOCATOR | CAST FROM datatype [AS LOCATOR]} ]
FOR udt_name
EXTERNAL [NAME 'name'] [TRANSFORM GROUP group_name]
code_body
```

The syntax for *ALTER METHOD* is quite a bit simpler:

```
ALTER METHOD method_name
FOR udt_name
EXTERNAL NAME 'name'
```

where:

[SPECIFIC] METHOD *method_name*

Defines the method being created or altered. The optional *SPECIFIC* keyword identifies a method using the name assigned (either implicitly or explicitly) using the *CREATE TYPE* statement. The method must exist in the named or current schema.

CAST FROM *datatype*

Returns a different datatype to the invoking statement from the datatype that was returned by the method.

EXTERNAL [NAME '*external_name*'] [TRANSFORM GROUP *group_name*]

Indicates that the statement is registering an external method, rather than creating an internal (SQL) method. The user-defined type created using the *CREATE TYPE* statement must be defined with a *LANGUAGE* other than SQL.

NAME '*name*'

Defines the name of the external code that implements the method. The name may be up to 254 characters long. The name must be appropriate for the type of *LANGUAGE* associated with the user-defined type. This clause may also define the name of an internal (SQL) method specification for user-defined types defined with *LANGUAGE SQL*.

TRANSFORM GROUP *group_name*

Specifies a transform group for user-defined structure type transformations when the method is invoked. When omitted, the transform group *DB2_FUNCTION* is the default.

The method specification is determined by the *LANGUAGE* setting of the *CREATE TYPE* statement associated with the method. For *LANGUAGE SQL*, the method specification is SQL. For any other *LANGUAGE* setting, the method specification is external.

The following example method, called **office_zip**, checks to see if a ZIP Code is valid based on the user-defined type **address_type**:

```
CREATE METHOD office (address address_type)
```

```
RETURNS INTEGER
```

```
FOR address_type
```

```
RETURN
```

```
(CASE
```

```
  WHEN (self.zip = address.zip) THEN 1 -- valid
```

```
  ELSE 0 -- invalid
```

```
END);
```

In the next example, the method **distance** uses an external routine and transform group to process the between two **address_types**:

```
CREATE METHOD distance (address_type)
```

```
FOR address_type
```

```
EXTERNAL NAME 'addrlib!distance'
```

```
TRANSFORM GROUP method_group;
```

Later, we could change the method so that it uses a different external routine:

```
ALTER METHOD distance ( )
```

```
FOR TYPE address_type
```

```
EXTERNAL NAME 'newaddrlib!distance_2';
```

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

CREATE/ALTER TYPE

CREATE ROLE Statement

CREATE ROLE allows the creation of a named set of privileges that may be assigned to users of a database. When a role is granted to a user, that user gets all the privileges and permissions of that role. Roles are generally accepted as one of the best means for maintaining security and controlling privileges within a database.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
CREATE ROLE role_name [WITH ADMIN {CURRENT_USER | CURRENT_ROLE}]
```

Keywords

```
CREATE ROLE role_name
```

Creates a new role and differentiates that role from a host DBMS user and other roles. A role can be assigned any permission that a user can be assigned. The important difference is that a role can then be assigned to one or more users, thus giving them all the permissions of that role.

```
WITH ADMIN {CURRENT_USER | CURRENT_ROLE}
```

Assigns a role immediately to the currently active user or currently active role along with the privilege to pass the use of the role on to other users. By default, the statement defaults to *WITH ADMIN CURRENT_USER*.

Rules at a Glance

Using roles for database security can greatly ease administration and user maintenance. The general steps for using roles in database security are:

1. Assess the needs for roles and pick the role names (e.g., *administrator*, *manager*, *data_entry*, *report_writer*, etc.).
2. Assign permissions to each role as if it were a standard database user using the *GRANT* statement. For example, the *manager* role might have permissions to read and write from all user tables in a database, while the *report_writer* role only has permissions to execute read-only reports against certain reporting tables in the database.
3. Use the *GRANT* statement to grant roles to users of the system according to the types of work they will perform.

Permissions can be disabled using the *REVOKE* command.

Programming Tips and Gotchas

The main problem with roles is that occasionally a database administrator will provide redundant permissions to a role and separately to a user. If you ever need to prevent a user's access to the resource in situations like this, you usually will need to *REVOKE* the permissions twice. The role must be revoked from the user and then the specific user-level privilege must also be revoked from the user.

DB2

Not supported.

Instead, DB2 uses operating system groups, so you can achieve similar functionality but you need to:

1. Create an OS group.
2. Grant or revoke permissions to the group.
3. Add users to the group at the OS level.

DB2 allows OS groups to be assigned special admin roles at the instance level using the **sysadm_group**, **sysctrl_group**, and **sysmaint_group** configuration parameters.

MySQL

Not supported.

Oracle

Although it is not currently supported by the ANSI standard, Oracle also offers an *ALTER ROLE* statement. Oracle supports the concept of roles, though Oracle's implementation of the commands is very different from the ANSI SQL standard:

```
{CREATE | ALTER} ROLE role_name  
  
[NOT IDENTIFIED |  
  
IDENTIFIED {BY password | EXTERNALLY | GLOBALLY |  
  
USING package_name}]
```

where:

```
CREATE | ALTER ROLE role_name
```

Identifies the name of the new role being created or the pre-existing role being modified.

```
NOT IDENTIFIED
```

Declares that no password is needed for the role to receive authorization to the database. This is the default.

```
IDENTIFIED
```

Declares that the users of the role must authenticate themselves by the method indicated before the role is enabled via the *SET ROLE* command, where:

```
BY password
```

Creates a local role authenticated by the string value of *password*. Only single-byte characters are allowed in the password even when using a multibyte character set.

```
EXTERNALLY
```

Creates an external role that is authenticated by the operating system or a third-party provider. In either case, the external authentication service will likely require a password.

GLOBALLY

Creates a global role that is authenticated by an enterprise directory service, such as an LDAP directory.

USING *package_name*

Creates an application role that enables a role only through an application that uses a PL/SQL package of *package_name*. If you omit the schema, Oracle assumes that the package is in your schema.

In Oracle, the role is created first, then granted privileges and permissions as if it is a user via the *GRANT* command. When users want to get access to the permissions of a role protected by a password, they use the *SET ROLE* command. If a password is placed on the role, any user wishing to access it must provide the password with the *SET ROLE* command.

Oracle ships with several preconfigured roles. *CONNECT*, *DBA*, and *RESOURCE* are available in all versions of Oracle. *EXP_FULL_DATABASE* and *IMP_FULL_DATABASE* are newer roles used for import and export operations. The *GRANT* statement reference has a more detailed discussion of all of the preconfigured roles available in Oracle.

The following example uses *CREATE* to specify a new role in Oracle, *GRANTS* it privileges, assigns it a password with *ALTER ROLE*, and *GRANTS* that role to a couple of users:

```
CREATE ROLE boss;
```

```
GRANT ALL ON employee TO boss;
```

```
GRANT CREATE SESSION, CREATE DATABASE LINK TO boss;
```

```
ALTER ROLE boss IDENTIFIED BY le_grande_fromage;
```

```
GRANT boss TO emily, jake;
```

PostgreSQL

Although PostgreSQL does not support the *CREATE ROLE* command, it offers a nearly identical facility of its own called *CREATE GROUP*. PostgreSQL also offers an *ALTER GROUP* statement to modify existing groups. In PostgreSQL, a group and a role are identical:

```
{CREATE | ALTER} GROUP name
```

```
[ [WITH] {SYSID int | USER username [,...]} ]
```

```
[ {DROP | ADD} username [,...]} ]
```

where:

```
{CREATE | ALTER} GROUP name
```

Specifies if a new group (i.e., role) is created or if an existing group is modified where *name* is the name of the group to create or modify.

```
WITH {SYSID int | USER username [,...]} ]
```

Assigns an explicit group ID to the new group, with the *SYSID* clause, or assigns a list of one or more pre-existing users to the group. When omitted, PostgreSQL will create the group using the highest assigned group ID plus one (starting at 1). The *WITH* clause is only used with the *CREATE GROUP* statement.

```
{DROP | ADD} username
```

Drops or adds one or more pre-existing users to the group. Used only with the *ALTER GROUP* clause.

Use the *DROP GROUP* clause to drop a group you no longer want.

SQL Server

Microsoft SQL Server does not support the *CREATE ROLE* command, but has the equivalent capability via the system stored procedure **sp_add_role**.

See Also

GRANT
REVOKE

CREATE SCHEMA Statement

This statement creates a *schema*—i.e., a named group of related objects. A schema is a collection of tables, views, and permissions granted to specific users or roles. According to the ANSI standard, specific object permissions are not schema objects in themselves and do not belong to a specific schema. However, roles are sets of privileges that do belong to a schema.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

CREATE SCHEMA [*schema_name*] [AUTHORIZATION *owner_name*]

[DEFAULT CHARACTER SET *char_set_name*]

[PATH *schema_name* [,...]]

[ANSI CREATE *statements* [...]]

[ANSI GRANT *statements* [...]]

Keywords

CREATE SCHEMA [*schema_name*]

Creates a schema called *schema_name*. When omitted, the database will create a schema name for you using the name of the user who owns the schema.

AUTHORIZATION *owner_name*

Sets the *owner_name* as the user who owns the schema. When omitted, the current user is set as the owner. The ANSI standard allows you to omit either the *schema_name* or the *AUTHORIZATION* clause, or to use them both together.

DEFAULT CHARACTER SET *char_set_name*

Declares a default character set of *char_set_name* for all objects created within the schema.

PATH *schema_name* [,...]

Optionally declares a file path and filename for any unqualified routines (i.e., stored procedures, user-defined functions, user-defined methods) in the schema.

ANSI CREATE *statements* [...]

Contains one or more *CREATE* statements. No commas are used between the *CREATE* statements.

ANSI GRANT *statements* [...]

Contains one or more *GRANT* statements to previously defined objects. Usually, the objects were created earlier in the same *CREATE SCHEMA* statement, but they may be any previously created object. No commas are used between the *GRANT* statements.

Rules at a Glance

The *CREATE SCHEMA* statement is a container that can hold many other *CREATE* and *GRANT* statements. The most common way to think of a schema is as all of the objects that a specific user owns. For example, the user **jake** may own several tables and views including the table, **publishers**, in his own schema. Meanwhile, the user **dylan** may own several other tables and views in his schema, but may also own his own separate copy of the **publishers** table.

The ANSI standard requires that all *CREATE* statements are allowed in a *CREATE SCHEMA* statement. In practice, most implementations of *CREATE SCHEMA* only allow three subordinate statements: *CREATE TABLE*, *CREATE VIEW*, and *GRANT*. The order of the commands is not important, meaning you can grant privileges on tables or views whose *CREATE* statements appear later in the *CREATE SCHEMA* statement.

Programming Tips and Gotchas

It is not required, but is considered a best practice, to arrange the objects and grants within a *CREATE SCHEMA* statement in an order that will execute naturally. In other words, *CREATE VIEW* statements should follow the *CREATE TABLE* statements that they depend on, and then *GRANT* statements should come last.

If your database system uses schemas, we recommend that you always reference your objects by schema and then object name (as in **jake.publishers**). If you do not include a schema qualifier, the database platform will typically assume the default schema for the current user connection.

Some database platforms do not explicitly support the *CREATE SCHEMA* command. However, they implicitly create a schema when a user creates database objects. For example, Oracle creates a schema whenever a user is created. The *CREATE SCHEMA* command is simply a single-step method of creating all the tables, views, and other database objects along with their permissions.

DB2

DB2 supports the basic elements of the *CREATE SCHEMA* statement. It *does not* support the *PATH* or *DEFAULT CHARACTER SET* clauses:

```
CREATE SCHEMA [schema_name] [AUTHORIZATION owner_name]
```

```
[ ANSI CREATE statements [...] ]
```

```
[ ANSI GRANT statements [...] ]
```

The syntax and usage is otherwise identical to the ANSI standard.

DB2 creates a schema implicitly whenever a new schema is referenced in a *GRANT* or *CREATE* statement.

DB2 does not support an *ALTER SCHEMA* statement.

MySQL

Not supported.

Oracle

In Oracle, the *CREATE SCHEMA* does not create a schema—only the *CREATE USER* statement does that. *CREATE SCHEMA* allows a user to perform multiple *CREATE*s and *GRANT*s in a previously created schema in one SQL statement:

```
CREATE SCHEMA AUTHORIZATION schema_name
```

[ANSI CREATE *statements* [...]]

[ANSI GRANT *statements* [...]]

Note that Oracle only allows ANSI standard *CREATE TABLE*, *CREATE VIEW*, and *GRANT* statements within a *CREATE SCHEMA* statement. You *should not* use any of Oracle's extensions to these commands when using the *CREATE SCHEMA* statement.

The following Oracle example places the permissions before the objects within the *CREATE SCHEMA* statement:

```
CREATE SCHEMA AUTHORIZATION emily

GRANT SELECT, INSERT ON view_1 TO sarah

GRANT ALL ON table_1 TO sarah

CREATE VIEW view_1 AS

    SELECT column_1, column_2

    FROM table_1

    ORDER BY column_2

CREATE TABLE table_1(column_1 INT, column_2 CHAR(20));
```

As the example above shows, the order of the statements within the *CREATE SCHEMA* statement is unimportant. Oracle commits the *CREATE SCHEMA* statement only if all *CREATE* and *GRANT* statements in the statement complete successfully.

PostgreSQL

Not supported.

SQL Server

SQL Server supports the basic *CREATE SCHEMA* statement without support for the *PATH* clause or the *DEFAULT CHARACTER SET* clause:

```
CREATE SCHEMA AUTHORIZATION owner_name
```

[ANSI CREATE *statements* [...]]

[ANSI GRANT *statements* [...]]

If any statement fails within the *CREATE SCHEMA* statement, then the entire statement fails.

SQL Server does not require that the *CREATE* or *GRANT* statements be in any particular order, except that nested views must be declared in their logical order. Thus, if **view_100** references **view_10**, then **view_10** must appear in the *CREATE SCHEMA* statement before **view_100**.

For example:

```
CREATE SCHEMA AUTHORIZATION katie

GRANT SELECT ON view_10 TO public

CREATE VIEW view_10(col1) AS SELECT col1 FROM foo

CREATE TABLE foo(col1 INT)

CREATE TABLE foo

    (col1 INT PRIMARY KEY,

    col2 INT REFERENCES foo2(col1))
```

```
CREATE TABLE foo2
(
  col1 INT PRIMARY KEY,
  col2 INT REFERENCES foo(col1))
```

See Also

CREATE/ALTER TABLE
CREATE/ALTER VIEW
GRANT

CREATE/ALTER TABLE Statement

Manipulating tables is one of the most common activities that database administrators and programmers perform when working with database objects. This section details how to create and modify tables.

The ANSI standard represents a sort of least common denominator among the vendors, though not every option of the ANSI standard version of *CREATE TABLE* and *ALTER TABLE* is offered by each of the vendors. However, the ANSI standard does represent the basic form that can be used across all of the platforms. In turn, the vendor platforms offer a variety of extensions and additions to the ANSI standards for *CREATE* and *ALTER TABLE*.



Typically, a great deal of consideration goes into the design and creation of a table. This discipline is known as *database design*. The discipline of analyzing the relationship of a table to its own data and to other tables within the database is known as *normalization*. We recommend that database programmers and administrators alike study both database design and normalization principles thoroughly before issuing *CREATE TABLE* commands.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 statement *CREATE TABLE* creates a permanent or temporary table within the database where the command is issued. The syntax is as follows:

```
CREATE [{LOCAL TEMPORARY} | GLOBAL TEMPORARY} TABLE table_name
```

```
(column_name datatype attributes [,...] ) |
```

```
[column_name WITH OPTIONS options] |
```

```
[LIKE table_name] |
```

```
[REF IS column_name
```

```
{SYSTEM GENERATED | USER GENERATED | DERIVED} ]
```

```
[CONSTRAINT constraint_type [constraint_name] [,...] ]
```

```
[OF type_name [UNDER super_table] [table_definition] ]
```

[ON COMMIT {PRESERVE ROWS | DELETE ROWS}]

The SQL2003 statement *ALTER TABLE* allows many useful modifications to be made to an existing table without dropping any existing indexes, triggers, or permissions assigned to it. Following is the *ALTER TABLE* syntax:

```
ALTER TABLE table_name  
[ADD [COLUMN] column_name datatype attributes]  
| [ALTER [COLUMN] column_name SET DEFAULT default_value]  
| [ALTER [COLUMN] column_name DROP DEFAULT]  
| [ALTER [COLUMN] column_name ADD SCOPE table_name]  
| [ALTER [COLUMN] column_name DROP SCOPE {RESTRICT | CASCADE}]  
| [DROP [COLUMN] column_name {RESTRICT | CASCADE}]  
| [ADD table_constraint]  
| [DROP CONSTRAINT table_constraint_name {RESTRICT | CASCADE}]
```

Keywords

[LOCAL | GLOBAL] TEMPORARY

Declares a permanent table or a *TEMPORARY* table of *LOCAL* or *GLOBAL* scope. *Local temporary tables* are only accessible to the session that created them and are automatically dropped when the session that created them terminates. *Global temporary tables* are accessible by all active sessions, but are automatically dropped when the session that created them terminates. Do not qualify a temporary table with a schema name.

(*column_name datatype attributes* [,...])

Defines a comma-delimited list of one or more columns, their datatypes, and any additional attributes such as nullability. Every table declaration must contain at least one column, which may include:

Column_name

Specifies a name for a column. The name must be a valid identifier according to the rules of the specific DBMS. Make sure the name makes sense!

Datatype

Associates a specific datatype with the column of *column_name*. An optional *length* may be specified for datatypes that allow user-defined lengths, for example *VARCHAR(255)*. It must be a valid datatype on the specific DBMS. Refer to [Chapter 2](#) for a full discussion of acceptable datatypes and vendor variations.

Attributes

Associates specific constraint attributes with the *column_name*. Many attributes may be applied to a single *column_name*, no commas required. Typical ANSI attributes include:

NOT NULL

Tells the column to reject null values or accept null values, when *NOT NULL* is omitted. Any *INSERT* or *UPDATE* statement that attempts to place a null value in a *NOT NULL* column will fail and roll back.

DEFAULT expression

Tells the column to use the value of *expression* when no value is supplied by an *INSERT* or *UPDATE* statement. The expression must be acceptable according to the datatype of the column; for example, no alphabetic characters may be used in an *INTEGER* column. *Expression* may be a string or numeric literal, but you may also define a user-defined function or system function. SQL2003 allows these system functions in a *DEFAULT* expression: *NULL*, *USER*, *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *LOCALTIME*, *CURRENT_TIMESTAMP*, *LOCALTIMESTAMP*, *ARRAY*, or *ARRAY[]*.

COLLATE collation_name

Defines a specific collation, or sort order, for the column it is assigned to. The name of the collation is platform dependent. If you do not define a collation, the columns of the table default to the collation of the character set used for the column.

REFERENCES ARE [NOT] CHECKED [ON DELETE {RESTRICT | SET NULL}]

Indicates whether references are to be checked on a *REF* column defined with a scope clause. The optional *ON DELETE* clause tells whether any values in records referenced by a deleted record should set to *NULL* or whether the operation should be restricted.

CONSTRAINT constraint_name [constraint_type [constraint]]

Assigns a constraint and, optionally a constraint name, to the specific column. Constraint types are discussed in [Chapter 2](#). Because the constraint is associated with a specific column, the constraint declaration assumes that the column is the only one in the constraint. After the table is created, the constraint is treated as a table-level constraint.

column_name [WITH OPTIONS options]

Defines a column with special options, such as a scope clause, a default clause, a column-level constraint, or a *COLLATE* clause. For many implementations, the *WITH OPTIONS* clause is restricted to the creation of typed tables.

LIKE table_name

Creates a new table with the same column definitions as the pre-existing table of *table_name*.

REF IS column_name {SYSTEM GENERATED | USER GENERATED | DERIVED}

Defines the object identifier column (OID) for a typed table. An OID is necessary for the root table of a table hierarchy. Based on the option, the *REF* might be automatically generated by the system (*SYSTEM GENERATED*), manually provided by the user when inserting the row (*USER GENERATED*), or derived from another *REF* (*DERIVED*). Requires the inclusion of a *REFERENCES* column attribute for *column_name*.

CONSTRAINT constraint_type [constraint_name] [,...]

Assigns one or more constraints to the table. This option is noticeably different than the *CONSTRAINT* option at the column level because column-level constraints are assumed to apply only to the column they are associated with. Table-level constraints, however, give the option of associating multiple columns with a constraint. For example, in a sales table you might wish to declare a unique constraint on a concatenated key of **store_id**, **order_id**, and **order_date**. This can only be done using a table-level constraint. Refer to [Chapter 2](#) for a full discussion of constraints.

OF type_name [UNDER super_table] [table_definition]

Declares that the table is based upon a pre-existing user-defined type (see [CREATE/ALTER TYPE Statement](#)). In this situation, the table may have only a single column for each attribute of the structured type, plus an additional column as defined in the *REF IS* clause. The *REF* datatype is described in greater detail in the section [CREATE/ALTER TYPE Statement](#). This clause is incompatible with the *LIKE table_name* clause.

where:

UNDER super_table [table_definition]

Declares the direct supertable of the currently declared table within the same schema, if any exists. The table being created is thus a direct subtable of the supertable. You may optionally provide a complete *table_definition* of the new subtable, complete with columns, constraints, etc.

ON COMMIT {PRESERVE ROWS | DELETE ROWS}

ON COMMIT PRESERVE ROWS preserves data rows in a temporary table on issuance of a *COMMIT* statement.
ON COMMIT DELETE ROWS deletes all data rows in a temporary table on *COMMIT*.

ADD [COLUMN] *column_name datatype attributes*

Adds a column to a table, along with the appropriate datatype and attributes.

ALTER [COLUMN] *column_name* SET DEFAULT *default_value*

Adds a default value to the column if none exists, and resets the default value if a previous one exists.

ALTER [COLUMN] *column_name* DROP DEFAULT

Completely removes a default from the named column.

ALTER [COLUMN] *column_name* ADD SCOPE *table_name*

Adds a *scope* to the named column. A scope is a reference to a user-defined datatype (see [CREATE/ALTER TYPE Statement](#)”).

ALTER [COLUMN] *column_name* DROP SCOPE [RESTRICT | CASCADE]

Drops a scope from the named column. *RESTRICT* and *CASCADE* clauses are defined at the end of this list.

DROP COLUMN *column_name* [RESTRICT | CASCADE]

Drops the named column from the table. *RESTRICT* and *CASCADE* clauses are defined at the end of this list.

ADD *table_constraint*

Adds a table constraint of the specified name and characteristics to the table.

DROP CONSTRAINT *constraint_name* [RESTRICT | CASCADE]

Drops a previously defined constraint from the table.

RESTRICT

Tells the DBMS to abort the command if it finds any other objects in the database that depend on the object.

CASCADE

Tells the DBMS to drop any other objects that depend on the object.

Rules at a Glance

The typical *CREATE TABLE* statement is very simple. Generally, it names the table and any columns contained in the table. Many table definitions also include a nullability constraint for most of the columns, as in this SQL Server example:

CREATE TABLE housing_construction

(project_number INT NOT NULL,

project_date DATE NOT NULL,

```
project_name    VARCHAR(50) NOT NULL,  
construction_color  NCHAR(20) ,  
construction_height DECIMAL(4,1),  
construction_length DECIMAL(4,1),  
construction_width  DECIMAL(4,1),  
construction_volume INT      )
```

This example adds a foreign key to the example table:

```
-- Creating a column-level constraint
```

```
CREATE TABLE favorite_books  
(isbn      CHAR(100) PRIMARY KEY,  
book_name  VARCHAR(40) UNIQUE,  
category   VARCHAR(40) ,  
subcategory VARCHAR(40) ,  
pub_date   DATETIME  NOT NULL,  
purchase_date DATETIME  NOT NULL,  
          CONSTRAINT fk_categories FOREIGN KEY (category)  
          REFERENCES category(cat_name));
```

The foreign key on the **categories** column relates it to the **cat_name** table in the **category** table. All the vendors mentioned in this book support this syntax.



Examples for creating a table with each constraint type are shown in [Chapter 2](#). Similarly, the foreign key could be added after the fact as a multicolumn key including both the **category** and **subcategory** columns:

```
ALTER TABLE favorite_books ADD CONSTRAINT fk_categories  
          FOREIGN KEY (category, subcategory)  
          REFERENCES category(cat_name, subcat_name));
```

Now, we could use an *ALTER TABLE* statement to drop the constraint altogether:

```
ALTER TABLE favorite_books DROP CONSTRAINT fk_categories RESTRICT;
```

Listed here are more full examples from **pubs**, the sample database that ships with Microsoft SQL Server and Sybase Adaptive Server:

```
-- For a Microsoft SQL Server database
```

```
CREATE TABLE jobs  
(job_id SMALLINT IDENTITY(1,1) PRIMARY KEY CLUSTERED,  
job_desc VARCHAR(50) NOT NULL DEFAULT 'New Position',  
min_lvl TINYINT NOT NULL CHECK (min_lvl >= 10),
```

```
max_lvl TINYINT NOT NULL CHECK (max_lvl <= 250))
-- For a MySQL database
CREATE TABLE employee
(emp_id INT AUTO_INCREMENT CONSTRAINT PK_emp_id PRIMARY KEY,
fname VARCHAR(20) NOT NULL,
minit CHAR(1) NULL,
lname VARCHAR(30) NOT NULL,
job_id SMALLINT NOT NULL DEFAULT 1
REFERENCES jobs(job_id),
job_lvl TINYINT DEFAULT 10,
pub_id CHAR(4) NOT NULL DEFAULT ('9952')
REFERENCES publishers(pub_id),
hire_date DATETIME NOT NULL DEFAULT (CURRENT_DATE( ));
CREATE TABLE publishers
(pub_id char(4) NOT NULL
CONSTRAINT UPKCL_pubind PRIMARY KEY CLUSTERED
CHECK (pub_id IN ('1389', '0736', '0877', '1622', '1756')
OR pub_id LIKE '99[0-9][0-9]'),
pub_name varchar(40) NULL,
city varchar(20) NULL,
state char(2) NULL,
country varchar(30) NULL DEFAULT('USA'))
```

Once you get into the vendor extensions, the *CREATE TABLE* command is no longer portable between database platforms. The following is an example of an Oracle *CREATE TABLE* statement with many storage properties:

```
CREATE TABLE classical_music_cds
(music_id INT,
composition VARCHAR2(50),
composer VARCHAR2(50),
performer VARCHAR2(50),
performance_date DATE DEFAULT SYSDATE,
duration INT,
cd_name VARCHAR2(100),
CONSTRAINT pk_class_cds PRIMARY KEY (music_id)
USING INDEX TABLESPACE index_ts
STORAGE (INITIAL 100K NEXT 20K),
CONSTRAINT uq_class_cds UNIQUE
(composition, performer, performance_date)
```

```
    USING INDEX TABLESPACE index_ts  
  
    STORAGE (INITIAL 100K NEXT 20K)  
  
TABLESPACE tabledata_ts;
```

When issuing a *CREATE* or *ALTER* statement, we recommend that it be the only statement in your transaction. For example, do not attempt to create a table and select from it in the same batch. Instead, (1) create the table, (2) verify the operation, (3) *COMMIT*, and then (4) perform any subsequent operations against the table.

Table_name is the name for a new or existing table. New table names should start with an alphabetic character and in general contain no other special symbol besides underscore (`_`). Length of the name and exact composition is governed differently according to each vendor.

When creating or altering a table, the list of column definitions is always encapsulated within parentheses and separated by commas between each new column definition.

Programming Tips and Gotchas

The user issuing the *CREATE TABLE* command must have the appropriate permissions. Similarly, any user wishing to *ALTER* or *DROP* a table should own the table within his own schema or have adequate permissions to the alter or drop the table. Since the ANSI standard does not specify the privileges required, expect some variation between vendors.

You can encapsulate a *CREATE TABLE* or *ALTER TABLE* statement within a transaction, using a *COMMIT* or *ROLLBACK* statement to explicitly conclude the transaction. We recommend that it be the *only* command in the transaction.

You can have a great deal of control over the way that the records of a table are physically written to disk through extensions to the ANSI standard. DB2 and SQL Server use a technique called *clustered indexes* to control the way that records are written to disk. Oracle uses a technique that is functionally similar called an *index-organized table*.

Some databases will lock a table that is being modified by an *ALTER TABLE* statement. It is wise to issue this command only on tables that are not in use on a production database.

Some databases will lock the target *and* source tables when using the *CREATE TABLE...LIKE* statement.

DB2

DB2 supports a very rich and elaborate set of extensions to the *CREATE TABLE* statement:

```
CREATE TABLE table_name  
  
    {(column_name datatype attributes [,...]) } |  
  
    OF type_name [{UNDER super_table INHERIT SELECT PRIVILEGES |  
    HIERARCHY hierarchy_name}] [table_definition] |  
  
    AS subquery {derived_table_options |  
    DEFINITION ONLY [copy_options] } |  
  
    [LIKE table_name] [copy_options] |  
  
    [ [(staging_column [,...]) ]  
    FOR table_name PROPAGATE IMMEDIATE] }  
  
[ORGANIZE BY [DIMENSIONS] (column_name [,...]) ]  
  
[DATA CAPTURE {NONE | CHANGES} ]  
  
[IN tablespace_name [INDEX IN tablespace_name]  
    [LONG IN tablespace_name]]  
  
[VALUE COMPRESSION]  
  
[WITH RESTRICT ON DROP]  
  
[NOT LOGGED INITIALLY]  
  
[{{REPLICATED | PARTITIONING KEY (column_name [,...])
```

```
[USING HASHING] } ]  
[OPTIONS (REMOTE_SERVER 'server_name'  
[, REMOTE_SCHEMA 'schema_name'  
[, REMOTE_TABNAME 'table_name' ] ]
```

Similarly rich, the DB2 statement *ALTER TABLE* allows many useful modifications to be made to an existing table without dropping any existing indexes, triggers, or permissions assigned to it:

```
ALTER TABLE table_name  
{[ADD { {[COLUMN] column_name datatype attributes } |  
  table_constraint | partitioning_key | RESTRICT_ON_DROP } |  
{[ALTER { {CHECK | FOREIGN KEY} constraint_name constraint_definition } |  
  [COLUMN] column_name column_definition } |  
{[DROP constraint_type constraint_name] | partitioning_key |  
  RESTRICT_ON_DROP } |  
[DATA CAPTURE {NONE | CHANGES [INCLUDE LONGVAR COLUMNS]} |  
[ACTIVATE NOT LOGGED INITIALLY [WITH EMPTY TABLE] ] |  
[PCTFREE int] |  
[LOCKSIZE {ROW | TABLE} ] |  
[APPEND {ON | OFF} ] |  
[ [NOT] VOLATILE [CARDINATITY] ] |  
[ [DE]ACTIVATE VALUE COMPRESSION] } [...] |  
SET MATERIALIZED QUERY AS {DEFINITION ONLY | subquery}
```

The parameters for both commands are as follows:

(*column_name datatype attributes* [,...])

Defines columns according to the ANSI standard. However, DB2 supports several additional attributes that are specific to special datatypes on DB2. All the ANSI standard attributes, such as *NOT NULL*, are supported. Refer to the section on datatypes in [Chapter 2](#) for more details.

UNDER *super_table* INHERIT SELECT PRIVILEGES | HIERARCHY *hierarchy_name*

Similar to the ANSI standard *OF* clause in the *CREATE TABLE* command, with two added optional clauses — *INHERIT* and *HIERARCHY*. *INHERIT* tells DB2 that any user or group with read privileges on the *super_table* should also have read privileges on the new subtable. *HIERARCHY* associates the subtable with a table hierarchy, where *hierarchy_name* is the name of the root table of the hierarchy. When omitted, DB2 will generate its own *hierarchy_name*.

table_definition

Defines the elements of the typed table, including addition options for columns, an *OID* column on the root table, and table constraints.

AS *subquery*

Materializes the table from the results of a query. You may include a standard column definition list, but the columns must match the number and datatype of columns retrieved by the subquery.

derived_table_options

The full syntax for *derived_table_options* is:

DATA INITIALLY DEFERRED REFRESH {DEFERRED | IMMEDIATE}

[{ENABLE | DISABLE} QUERY OPTIMIZATION]

[MAINTAINED BY {SYSTEM | USER}]

The keywords have the following meanings:

DATA INITIALLY DEFERRED

Defers the insertion of data into the table until a DB2 *REFRESH TABLE* command is issued.

REFRESH {DEFERRED | IMMEDIATE}

Allows the table to be refreshed only when a DB2 *REFRESH TABLE* statement is issued (via *DEFERRED*) or immediate after any data manipulation operation (via *IMMEDIATE*).

{ENABLE | DISABLE} QUERY OPTIMIZATION

Enables or disables the derived table for query optimization when DB2 decides optimization is appropriate. Of course, the table is always available for direct queries.

MAINTAINED BY {SYSTEM | USER}

Declares that the derived table data is maintained automatically by the *SYSTEM* (the default) or manually by the *USER* via the DB2 *REFRESH TABLE* command. If you select *MAINTAINED BY USER*, the table must also be declared as *REFRESH DEFERRED*.

DEFINITION ONLY [*copy options*]

Defines the table without populating the table with a query result set.

copy_options

Allows the modification of the *DEFINITION ONLY* and *LIKE* clauses. Copy options include:

{INCLUDING | EXCLUDING} [COLUMN] DEFAULTS

Includes or excludes column defaults of the copied table, respectively. *INCLUDING* is the default for base tables or temporary tables. Otherwise, *EXCLUDING* is the default.

{INCLUDING | EXCLUDING} [IDENTITY] [COLUMN ATTRIBUTES]

Includes or excludes column attributes of the copied table, respectively. The *IDENTITY* attributes of the column may optionally be copied.

LIKE *table_name* [*copy options*]

Copies the entire set of column definitions from an existing table, *table_name*, though a view or nickname is also allowed. You may use the **copy_options** shown previously in the list to alter the default behavior of the *LIKE* clause.

[(*staging_column* [,...])] FOR *table_name* PROPAGATE IMMEDIATE]

Defines an intermediate table (i.e., a staging table) between the base table(s) that define a derived table and

the derived table itself. (Naturally, this clause should only be used with the *AS subquery* clause.)

The components of this clause are:

staging_column

Defines one or more staging columns. However, the list of staging columns will always contain two more columns than the number of columns in the derived table — *GLOBALTRANSID* and *GLOBALTRANSTIME*. The *PROPAGATE* clause, adds one more column — *OPERATIONTYPE* — which is used to maintain consistency in replicated a materialized query table. When omitted, DB2 will create the appropriate staging columns it needs for the operation.

FOR *table_name*

Names the table that defines the staging table. *table_name* must represent a derived table with *REFRESH DEFERRED* already existing in the database. Whenever a *REFRESH TABLE* command is issued, the contents of the staging table are used to refresh the derived table.

PROPAGATE IMMEDIATE

Specifies that data modifications to the underlying tables cascade immediately to the staging table.

ORGANIZE BY [DIMENSIONS] (*column_name* [,...])

Defines a cluster or dimension on a column or group of columns, depending on whether the *DIMENSIONS* keyword is excluded or included, respectively. In effect, a clustered index is defined with the *ORGANIZE BY* clause, while the *ORGANIZE BY DIMENSIONS* clause is used to create a clustered index on more than one column (i.e., on a dimension).

DATA CAPTURE {NONE | CHANGES}

Writes extra data to the log in support of inter-database replication. *NONE* declares that no extra information will be stored, while *CHANGES* declares that changes are written to the log. The *DATA CAPTURE CHANGES* clause is required on replicated tables. This clause is incompatible when creating subtables.

IN *tablespace_name* [INDEX IN *tablespace_name*][LONG IN *tablespace_name*]

Declares a specific, pre-existing tablespace where the table is placed. This clause is incompatible when creating a subtable and in System Managed tablespaces. The components of the clause are:

INDEX IN *tablespace_name*

Declares a separate pre-existing tablespace where the indexes of the table will be placed. When omitted, the default is the same tablespace as the table data.

LONG IN *tablespace_name*

Declares a separate pre-existing tablespace where the data of long datatypes (*LONG VARCHAR*, *LONG VARGRAPHIC*, *LOB*, etc.) for the table will be placed. When omitted, the default is the same tablespace as the table data.

VALUE COMPRESSION

Enables high compression storage of NULL and 0-length data values.

WITH RESTRICT ON DROP

Restricts the table and its tablespace from being dropped.

NOT LOGGED INITIALLY

Declares that any changes to the data of the table that occur in the same batch as when the table is created are not logged. On the other hand, all catalog changes and storage changes are logged. This setting can cause the rollback of an entire batch of work under certain circumstances.

REPLICATED

Declares that the data in the table is replicated on each database partition of the database partition group where it is defined, in effect copying the data to all database partitions. This clause is incompatible with *PARTITIONING KEY*, but is required for derived tables.

PARTITIONING KEY (*column_name* [,...]) [USING HASHING]

Defines a partitioning key for the table where *column_name* becomes the partitioning key. The *column_name* may not be a long or special datatype — *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK* — or structured or distinct types built upon these datatypes. This clause is incompatible when creating subtables. When omitted and the table is defined on a database partition group, DB2 will attempt to find a method for partitioning the table. Otherwise, the table will be placed on a single-partition database partition group. *USING HASHING* declares that a hashing algorithm is used to distribute data among the partitions.

OPTIONS (REMOTE_SERVER '*server_name*' [, REMOTE_SCHEMA '*schema_name*'] [, REMOTE_TABNAME '*table_name*']

Defines a remote operation, such as creating, altering, or dropping a table on a federated system. This is called *transparent DDL*. Federated systems are quite complex and are outside of the ANSI standard. Refer to the vendor documentation for more information.

ADD *table_constraint*

Adds a table-level constraint to an existing table.

ALTER {CHECK | FOREIGN KEY} *constraint_name constraint_definition*

Alters the attributes of an existing *CHECK* or *FOREIGN KEY* of *constraint_name* and assigns it a new *constraint_definition*. You may also apply these additional constraint characteristics:

[NOT] ENFORCED

Changes the constraint to *ENFORCED* or *NOT ENFORCED*. Under *ENFORCED*, the constraint is automatically enforced by the database. *NOT ENFORCED* causes the constraint to not be checked during data modification operations.

{ENABLE | DISABLE} QUERY OPTIMIZATION

Enables or disables query optimization on the constraint.

ALTER [COLUMN] *column_name column_definition*

Alters a specific column, *column_name*, in the table. You can change a number of column characteristics beyond the standard column definition options, including things like *SCOPE*, *COMPRESS*, and identity alterations. Refer to the material later in this section for information on type-related attributes and other alterable attributes. Some additional characteristics can be changed on a column-by-column basis, such as:

SET DATA TYPE VARCHAR (*int*)

Alters the length of the named *VARCHAR* column to *int*. The value of *int* should not increase the total row length or the total primary key length beyond the allowed maximum.

SET EXPRESSION AS (*generation-expression*)

Alters the original generated expression, a sort of auto-incrementing identity column, of the named column to a new *generation-expression*. The difference between a generated expression

and an identity is that identities are automatically handled by DB2, but generated expressions give you full control over the values returned by the expression. Generated expressions can be quite complex and are covered in the vendor documentation. This clause requires the table be placed in "check pending state," where the data in the table is not verified against constraints, via DB2's *SET INTEGRITY* statement. Afterwards, the values must all be checked.

SET INLINE LENGTH *int*

Alters the inline length of the named structured type column to *int*, storing the values inline with the rest of the data in the row.

DROP *constraint_type constraint_name*

Drops a constraint of *constraint_name* from the table. The *constraint_type* may be any of the allowable types: primary key, foreign key, unique, check, as well as *PARTITIONING KEY*. You may also substitute *constraint_type* with *RESTRICT ON DROP*, to drop the *RESTRICT ON DROP* characteristic.

DATA CAPTURE {*NONE* | *CHANGES* [*INCLUDE LONGVAR COLUMNS*] }

Captures extra information to the log in support of replication. Usable with any non-typed table. The options for *DATA CAPTURE* are:

NONE

Specifies that no extra information is captured.

CHANGES

Writes extra information to the table in support of replication. This clause is required for replication. This clause is incompatible with tables that store data on non-catalog partitions. The *INCLUDE LONGVAR COLUMNS* clause includes *LONG VARCHAR* and *LONG VARCHAR* columns in the data replication.

ACTIVATE NOT LOGGED INITIALLY [*WITH EMPTY TABLE*]

Disables logging of any transaction made in the same batch as the *ALTER TABLE* statements. Statements that are not logged include *INSERT*, *DELETE*, *UPDATE*, *CREATE INDEX*, *DROP INDEX*, or another *ALTER TABLE* statement. Logging returns to normal after the current SQL batch completes. The optional *WITH EMPTY TABLE* clause tells DB2 to empty the table of data. The data is not recoverable except through the DB2 *RECOVER* facility.

PCTFREE *int*

Declares a percentage of free space, *int*, that DB2 should leave on each page during load or reorganization of the table.

LOCKSIZE {*ROW* | *TABLE*}

Declares that locks should be taken at a *ROW* or *TABLE* level when reading or writing data in the table. The default is *ROW*. This clause is only supported in the root table of a typed-table hierarchy.

APPEND {*ON* | *OFF*}

Adds all new data to the end of the table when turned *ON* or puts data in the free space available within data pages throughout the table when turned *OFF*. The default is *OFF*. Setting this value to *ON* causes DB2 not to keep any free space information. The *APPEND* clause is not usable on tables with a clustered index.

[*NOT*] *VOLATILE* [*CARDINALITY*]

Declares whether the table cardinality varies greatly over time. When set to *VOLATILE*, tells DB2 to generally ignore statistics in favor of an index scan because the amount of data in the table change a great deal. *VOLATILE* is supported only on normal tables and the root table of a typed-table hierarchy.

NOT VOLATILE tells DB2 that the amount of data in the table is generally static, so that it can use existing statistics and optimization paths. Adding the optional *CARDINALITY* keyword tells DB2 that the amount of data in the table may change but that the table itself is static.

[DE]ACTIVATE VALUE COMPRESSION

Deactivates or activates the compression of NULL and zero-length data values in the table for all datatypes of *BLOB*, *CLOB*, *DBLOB*, *LONG VARCHAR*, or *LONG VARGRAPHIC*.

SET MATERIALIZED QUERY {AS DEFINITION ONLY | *subquery*}

Alters the properties of a materialized query table. *DEFINITION ONLY* tells DB2 to consider the table as a normal table and not a materialized query table. When using *DEFINITION ONLY*, the materialized query table cannot be replicated. The *subquery* clause is any valid subquery used to define a materialized query table. The *subquery* redefines a regular table as a materialized query table during query optimization. When using the *subquery* clause, the table may not be a typed table, have constraints, unique indexes, or triggers, or be defined as or referenced in a materialized query table.

DB2 also has a number of special attributes that may be applied to columns over the standard constraints that the ANSI SQL standard provides. These optional attributes are specific to specific datatypes. The following are *LOB*-related attributes:

[NOT] LOGGED

Declares whether changes made to the column are written to the logfile (*LOGGED*) or not (*NOT LOGGED*). The default is *LOGGED*. However, any *LOB* values greater than 10 megabytes should not be logged and those over 1 gigabyte cannot be logged.

[NOT] COMPACT

Compresses the values of *LOBs* as much as possible, usually by freeing any extra space on the last group used by the *LOB* or allowing free space in the last group to go unused. *COMPACT* takes some extra processing and I/O overhead. *NOT COMPACT* is the default.

For example, you might have a table called **emp_images**, which stores the security ID images of your employees:

```
CREATE TABLE emp_images
(emp_id      INT NOT NULL,
 emp_photo_image LONGVARGRAPHIC NOT LOGGED COMPACT)
```

DB2 supports the following datalink-related attributes:

LINKTYPE URL

Defines a URL as the type of link.

NO LINK CONTROL

Declares that there is no check for file existence.

FILE LINK CONTROL [*options*]

Declares that file existence will be checked. Additional options may impact the style of check, including:

INTEGRITY [ALL]

Sets the level of checks between the *DATALINK* value and an actual file. The optional *ALL* keyword disables operating system control for deleting or renaming the file.

READ PERMISSION {FS | DB}

Sets read permission on the file as either filesystem derived (*FS*) or database derived (*DB*).

WRITE PERMISSION {FS | BLOCKED | ADMIN}

Sets write permission on the file as either filesystem derived (*FS*), no writes allowed (*BLOCKED*), or determined by the Data Links Manager (*ADMIN*). The *WRITE PERMISSION ADMIN* clause is based on a passed token. You should further specify *[NOT] REQUIRING TOKEN FOR UPDATE*, which details whether a write token is needed to finish a file update operation.

RECOVERY {YES | NO}

Declares that DB2 will support point in time recovery of files referenced by the values in the column (*YES*) or not (*NO*). *RECOVERY YES* also requires the *INTEGRITY ALL* clause and either *WRITE PERMISSION BLOCKED* or *WRITE PERMISSION ADMIN*.

ON UNLINK {RESTORE | DELETE}

Declares that DB2 either return the file to its original permissions when the *DATALINK* value is changed or deleted (*RESTORE*) or that the file is deleted when unlinked (*DELETE*). *ON UNLINK RESTORE* requires the *INTEGRITY ALL* clause and either *WRITE PERMISSION BLOCKED* or *WRITE PERMISSION ADMIN*. *ON UNLINK DELETE* requires the *READ PERMISSION DB* clause and either *WRITE PERMISSION BLOCKED* or *WRITE PERMISSION ADMIN*.

MODE DB2OPTIONS

Defines a default group of file link options: *INTEGRITY ALL*, *READ PERMISSION FS*, *WRITE PERMISSION FS*, and *RECOVERY NO*.

DB2 supports the following type-related attributes:

SCOPE type_name

Declares the scope of a reference-type column, where *type_name* is the pre-existing typed table or view. It is required for columns that will be used as the left operand of a dereference operator or as the argument of a *DEREF* function.

INLINE LENGTH int

Declares a maximum byte size, *int*, for a structured-type column. If the structured type is too long, DB2 will automatically store the structured types in a separate table from the base table, similar to the way *LOB* values are stored.

DB2 supports just one storage-related attribute:

COMPRESS SYSTEM DEFAULT

Defines that the system default value for any column is compressed. It should not be used on columns of *DATE*, *TIME*, or *TIMESTAMP* datatypes. You gain some storage space at the sacrifice of performance.

Use the following *DEFAULT*-related attributes to set various defaults:

[WITH] DEFAULT {CURRENT SCHEMA | USER | NULL | constant}

Declares a default for the column. That is, whenever an *INSERT* or *UPDATE* (or *LOAD*) statement does not provide a value for the column, then DB2 will use the default value you specify. *WITH* is a noise word. When omitted, *NULL* is always the default. The options for the *DEFAULT* clause are:

CURRENT SCHEMA

Inserts the *CURRENT SCHEMA* system value. The column must have a character datatype of sufficient length to store *CURRENT SCHEMA*.

USER

Inserts the *USER* system value. The column must have a character datatype of sufficient length to store *USER*.

NULL

Inserts a null value, the default.

constant

Can be any constant string, date, or number of 254 characters or fewer.

GENERATED {ALWAYS | BY DEFAULT}

Tells DB2 to generate values for the column, in particular for *IDENTITY* columns, either every time a record is inserted in the table (*ALWAYS*) or whenever a value is not provided for the column (*BY DEFAULT*).

AS (expression)

Assigns the default value based on the evaluation of an *expression*. The expression might be something like an external user-defined function, a subquery, or a stored procedure. Refer to the *ALTER COLUMN...generated expressions* material earlier in this section.

AS IDENTITY [START WITH int | INCREMENT BY int]

Declares that the column is an auto-incrementing identity for a column of a number datatype. Identities can be either ascending or descending in nature. DB2 allows only one identity column per table. All identity columns are implicitly *NOT NULL*. Identities are further defined by these optional characteristics:

START WITH int

Declares the seed value (i.e., the starting value), *int*, of the identity column. The default is *MINVALUE* for ascending identities and *MAXVALUE* for descending identities.

INCREMENT BY int

Declares the amount each entry to the identity will increase (if positive) or decrease (if negative). The default is 1. A negative *INCREMENT BY* value tells DB2 that the identity is descending in nature.

[NO] {MINVALUE | MAXVALUE} [int]

Declares the value, *int*, at which the identity either cycles to the beginning or stops generating values. Both clauses may appear as attributes of an identity. They are not mutually exclusive. *NO MINVALUE*, the default minimum value setting, tells DB2 to start ascending identities at the *START WITH* value or 1, if omitted. *MINVALUE int* tells DB2 the lowest allowable value for a descending identity. *NO MAXVALUE*, the default maximum value setting, tells DB2 to stop or recycle ascending identities at maximum value allowed for the datatype. For descending identities, the default is the *START WITH* value or -1, when omitted. *MINVALUE int* tells DB2 the highest allowable value for an ascending identity.

[NO] CYCLE

Declares that DB2 should recycle the identity values (*CYCLE*) when the maximum value is exceeded on ascending identities (or the minimum value is decremented on a descending identity) or that DB2 should stop generating identities (*NO CYCLE*) once the minimum or maximum is reached. *NO CYCLE* is the default. DB2 will generate duplicate values under the *CYCLE* setting.

[NO] CACHE [int]

Tells DB2 to cache some pre-allocated values in memory (*CACHE*) or not (*NO CACHE*). You may optionally tell DB2 how many identity values, *int*, to cache using *CACHE int*.

[NO] ORDER

Tells DB2 whether identity values must be generated in the order they were requested (*ORDER*) or not (*NOT ORDER*). When omitted, *NOT ORDER* is the default.

For example, our company has acquired another company with a whole roster of employees. We want to assign the new **employees** a new **emp_id** based on our current highest **emp_id** identity column value:

```
CREATE TABLE temp_emp
(new_emp_id INT NOT NULL GENERATED ALWAYS AS IDENTITY
 (START WITH 500, INCREMENT BY 1, NO CYCLE, CACHE),
dept    VARCHAR (50) NOT NULL,
mgr_id  INT,
location CHAR(30))
```



DB2 supports the creation of temporary tables, but not in the *CREATE TABLE* statement. Instead, use the *DECLARE GLOBAL TEMPORARY TABLE* statement described in the DB2 documentation.

However, be extra careful whenever altering a table. You should generally expect that any procedural objects that are dependent on the table, such as procedures or functions, will become invalid.

-- Creating a column-level constraint

```
CREATE TABLE favorite_books
(isbn    CHAR(100) PRIMARY KEY,
book_name VARCHAR(40) UNIQUE,
category VARCHAR(40) NULL,
subcategory VARCHAR(40) NULL,
pub_date DATETIME NOT NULL,
purchase_date DATETIME NOT NULL,
CONSTRAINT fk_categories FOREIGN KEY (category)
REFERENCES category(cat_name));
```

In the following example, we alter our example table, **favorite_books**, to not log SQL changes for replication. We also assign some free space to the data pages of the table:

```
ALTER TABLE favorite_books
PCTFREE 20
DATA CAPTURE NONE
```

We also finally decide that although the records in the table change quite frequently, the table structure itself should be static:

```
ALTER TABLE favorite_books NO VOLATILE CARDINALITY
```

MySQL

The MySQL syntax for *CREATE TABLE* creates a permanent or local temporary table within the database in which the

command is issued.

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name
{(column_name datatype attributes
  constraint_type constraint_name [,...])
 [constraint_type [constraint_name] [,...]]
 [MATCH FULL | MATCH PARTIAL]
 [ON {DELETE | UPDATE} {RESTRICT | CASCADE | SET NULL |
  NO ACTION | SET DEFAULT}]
LIKE other_table_name }
[[TYPE = {ISAM | MyISAM | HEAP | BDB | InnoDB | MERGE |
MRG_MyISAM} |
  AUTO_INCREMENT = int |
  AVG_ROW_LENGTH = int |
  CHECKSUM = {0 | 1} |
  COMMENT = "string" |
  DELAY_KEY_WRITE = {0 | 1} |
  MAX_ROWS = int |
  MIN_ROWS = int |
  PACK_KEYS = {0 | 1} |
  PASSWORD = "string" |
  ROW_FORMAT= { default | dynamic | static | compressed } |
  RAID_TYPE = {1 | STRIPED | RAID0} |
  RAID_CHUNKS = int |
  RAID_CHUNKSIZE = int |
  INSERT_METHOD = {NO | FIRST | LAST} |
  DATA DIRECTORY = "path to directory"
  INDEX DIRECTORY = "path to directory" }
[[IGNORE | REPLACE] select_statement]
```

The MySQL syntax for *ALTER TABLE* allows modifications to a table or even renaming of a table:

```
ALTER [IGNORE] TABLE table_name
{[ADD [COLUMN] (column_name datatype attributes)
 [FIRST | AFTER column_name]]
 | [ADD [UNIQUE | FULLTEXT] INDEX [index_name]
 | (index_col_name [,...]) ]
 | [ADD PRIMARY KEY (index_col_name [,...]) ]
 | [ALTER [COLUMN] column_name {SET DEFAULT literal |
 | DROP DEFAULT}]
 | [CHANGE | MODIFY] [COLUMN] old_col_name new_column_name]
```

[FIRST | AFTER *column_name*]
| [MODIFY] *column_name datatype attributes*
| [DROP [COLUMN] *column_name*]
| [DROP PRIMARY KEY]
| [DROP INDEX *index_name*]
| [{ENABLE | DISABLE} KEYS]
| [RENAME [TO] *new_tbl_name*]
| [table_options] } [,...]

Keywords and parameters are as follows:

TEMPORARY

Creates a table that persists for the duration of the connection under which it was created. Once that connection closes, the temporary table is automatically deleted.

IF NOT EXISTS

Prevents an error if the table already exists. A schema specification is not required.

constraint_type

Allows standard ANSI SQL constraints to be assigned at the column or table level. MySQL fully supports these constraints: primary key, unique, and default (must be a constant). MySQL provides syntax support for check, foreign key, and references constraints, but they are not functional except on InnoDB tables. MySQL also has three specialty constraints:

FULLTEXT [INDEX]

Creates a fulltext-search index to facilitate rapid searches of large blocks of text. Note that only MyISAM tables support *FULLTEXT* indexes and that they can only be made on *CHAR*, *VARCHAR*, and *TEXT* columns.

AUTO_INCREMENT

Sets up an integer column so that it automatically increases its value by 1 (starting with a value of 1). MySQL only allows one *AUTO_INCREMENT* column per table. When the maximum value is deleted, the value is reused. When all records are deleted (using *DELETE* or *TRUNCATE*), the values may start over. This option is allowed on ISAM tables, but not MyISAM and **InnoDB** tables.

[UNIQUE] INDEX

When an *INDEX* characteristic is assigned to a column, a name for the index also can be included. (A MySQL synonym for *INDEX* is *KEY*.) If a name is not assigned to the primary key, MySQL assigns a name of *column_name* plus a numeric suffix (*_2*, *_3*, . . .) to make it unique. All table types, except for the ISAM table type, support indexes on NULL columns, or on *BLOB* or *TEXT* datatype columns. Note that *UNIQUE* without *INDEX* is valid syntax on MySQL.

TYPE

Describe how the data should be physically stored. You can convert tables between types using the *ALTER TABLE* statement. Only InnoDB and BDB tables offer recoverability with the *COMMIT* or *ROLLBACK* statements. All other table types will result in the loss of data if the database crashes, but are much faster and take up less space. The default is MyISAM. Following is the list of MySQL table types:

BDB

Creates a transaction-safe table with page locking. Tables of this type are based on the BerkeleyDB transaction processing engine available at <http://www.sleepycat.com>. BDB maintains logfiles and keeps track of checkpoints and *ROLLBACK* and *COMMIT* statements. BDB tables must have a declared primary key. Some operations on BDB tables are slower than the same operations on MyISAM tables.

HEAP

Creates a memory resident table that uses hashed indexes. Since they are memory resident, they are not transaction safe. Any data that they contain will be lost if the server crashes. Think of HEAP tables as an alternative to temporary tables. If you use HEAP tables, always specify the *MAX_ROWS* option so that you do not use all available memory. *HEAP* tables also do not support *BLOB* columns, *TEXT* columns, *AUTO_INCREMENT* constraints, *ORDER BY* clauses, or variable-length record format.

InnoDB

Creates a transaction-safe table with row-level locking. It also provides an independently managed tablespace, checkpoints, non-locking reads and fast reads from large datafiles for heightened concurrency and performance. Requires the *innodb_data_file_path* startup parameter. InnoDB supports all ANSI constraints including *CHECK* and *FOREIGN KEY*. (InnoDB is a separate complete transaction processing engine available at <http://www.innodb.com>. Well-known developer web sites like Slashdot.org run on InnoDB.) InnoDB requires special setup, including a possible recompile of the MySQL kernel, *init* file parameters, and the creation of tablespaces. Although InnoDB is included by default in the binary distributions of MySQL 4.0 and greater, you should check with your DBA to ensure you can use InnoDB tables.

ISAM

Creates a table in the older ISAM style. ISAM is deprecated as a feature and will probably not be supported beyond MySQL v5.0.

MERGE | MRG_MyISAM

Collects several identically structured MyISAM tables for use as one table. The *MERGE* table type provides some of the performance benefits of a partitioned table. The two keywords, *MERGE* and *MRG_MyISAM*, are synonyms. *SELECT*, *DELETE*, and *UPDATE* statements operate against the collection of tables as if they were one table. Think of a merge table as the collection, but not as the table(s) containing the data. Dropping a merge table only removes it from the collection. It does not erase the table or data from the database. Specify a *MERGE* table by using the statement *UNION=(table1, table2, ...)*.

MyISAM

Stores data in *.MYD* files and indexes in *.MYI* files. MyISAM is based on ISAM code with several extensions. MyISAM is a binary storage structure that is more portable than ISAM. MyISAM supports *AUTO_INCREMENT* columns, compressed tables (with the *mysampack* utility), and large table sizes. Under MyISAM, *BLOB* and *TEXT* columns can be indexes, and up to 32 indexes are allowed per table.

AUTO_INCREMENT = *int*

Sets the auto_increment value, *int*, for the table (MyISAM only).

AVG_ROW_LENGTH = *int*

Sets an approximate average row length for tables with variable-size records. MySQL uses *avg_row_length* * *max_rows* to decide how big a table may be.

CHECKSUM = {0 | 1}

When set to 1, maintains a checksum for all rows in the table (MyISAM only). Makes processing slower, but is less prone to corruption.

COMMENT = "*string*"

Allows a comment of up to 60 characters.

DELAY_KEY_WRITE = {0 | 1}

When set to 1, delays key table updates until the table is closed (MyISAM only).

MAX_ROWS = *int*

Sets a maximum number of rows to store in the table. The default maximum is 4 GB of space.

MIN_ROWS = *int*

Sets a minimum number of rows to store in the table.

PACK_KEYS = {0 | 1}

When set to 1, compacts the indexes of the table, making reads faster but updates slower (MyISAM and ISAM only). By default, only strings are packed. When set to 1, both strings and numeric values are packed.

PASSWORD = "*string*"

Encrypts the *.FRM* file with a password, "*string*", but not the table itself.

ROW_FORMAT = { *default* | *dynamic* | *static* | *compressed* }

Determines how future rows should be stored in a MyISAM table. *DYNAMIC* allows the rows to be of variable sizes (i.e., using *VARCHAR*) while *FIXED* expects fixed size columns (i.e., *CHAR*, *INT*, etc.).

RAID_TYPE = {1 | *STRIPED* | *RAID0*}

Enables tables to cross the 2 gigabyte/4 gigabyte size limit for MyISAM datafiles (but not index files) on operating systems that don't support very large files. The *RAID_TYPE* clause is intended as a workaround for database servers experiencing the aforementioned size limits, so you should not use *RAID_TYPE* on operating systems that support larger files. The keywords 1, *STRIPED*, and *RAIDZERO* are all synonyms, where *STRIPED* is the most commonly seen. The *--with-raid* init parameter is required. Currently, MySQL handles the *RAID_CHUNKS* and *RAID_CHUNKSIZE* clauses automatically.

RAID_CHUNKS = *int*

Overrides the default subdirectories used for the *RAID_TYPE*. MySQL usually creates subdirectories named 00, 01, and 02 in the database directory for all each MyISAM table. Each subdirectory will contain a file called *my_table_name.MYD*.

RAID_CHUNKSIZE = *int*

Sets a chunksize other than the default of 1,024 bytes. As the file size for the MyISAM table grows, MySQL will allocate additional space in *int* bytes in a round-robin fashion.

INSERT_METHOD = {*NO* | *FIRST* | *LAST*}

Required for *MERGE* tables. If no setting is specified for a *MERGE* table or the value is *NO*, then *INSERTS* are disallowed. *FIRST* inserts all records to the first table in the collection, while *LAST* inserts them all to the last table of the collection

DATA DIRECTORY = "*path to directory*"

Overrides the default path and directory that MySQL should use for MyISAM table storage.

INDEX DIRECTORY = "*path to directory*"

Overrides the default path and directory that MySQL should use for MyISAM index storage.

[IGNORE | REPLACE] *SELECT_statement*

Creates a table with columns based upon the elements listed in the *SELECT* statement. The new table will be populated with the results of the *SELECT* statement if the statement returns a result set.

ALTER [IGNORE]

The altered table will include all duplicate records unless the *IGNORE* keyword is used. Otherwise, the statement will fail if a duplicate row is encountered, the table has a unique index or primary key, and *IGNORE* is not used.

{ADD | COLUMN} [FIRST | AFTER *column_name*]

Adds or moves a column, index, or key to the table. When adding or moving columns, the new column appears as the last column on the table unless it is placed *AFTER* another named column.

ALTER COLUMN

Allows the definition or reset of a default value for a column. If you reset a default, MySQL will assign a new default value to the column.

CHANGE

Renames a column or changes its datatype.

MODIFY

Changes a column's datatype or attributes such as *NOT NULL*. Existing data in the column is automatically converted to the new datatype.

DROP

Drops the column, key, or index. A dropped column is also removed from any indexes it participated in. When dropping a primary key, MySQL will drop the first unique key if no primary key is present.

{ENABLE | DISABLE} KEYS

Enables or disables all non-unique keys on a MyISAM table simultaneously. This can be useful for bulk loads where you want to temporarily disable constraints until after the load is finished. It also speeds performance by finishing all index block flushing at the end of the operation.

RENAME [TO] *new_tbl_name*

Renames a table.

For example:

```
CREATE TABLE test_example
```

```
(column_a INT NOT NULL AUTO_INCREMENT,
```

```
PRIMARY KEY(column_a),
```

```
INDEX(column_b))
```

```
TYPE=HEAP
```

```
IGNORE
```

```
SELECT column_b,column_c FROM samples;
```

This creates a heap table with three columns: **column_a**, **column_b**, and **column_c**. Later on, we could change this

table to the MyISAM type:

```
ALTER TABLE example TYPE=MyISAM;
```

Three operating-system files are created when MySQL creates a MyISAM table: a table definition file with the extension *.FRM*, a datafile with the extension *.MYD*, and an index file with the extension *.MYI*. The *.FRM* datafile is used for all other tables.

The following example creates two base MyISAM tables and then creates a *MERGE* table from them:

```
CREATE TABLE message1
  (message_id INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20));
CREATE TABLE message2
  (message_id INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20));
CREATE TABLE all_messages
  (message_id INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20))
  TYPE=MERGE UNION=(message1, message2) INSERT_METHOD=LAST;
```

The following example renames both a table and a column:

```
ALTER TABLE employee RENAME AS emp;
ALTER TABLE employee CHANGE employee_ssn emp_ssn INTEGER;
```

Since MySQL allows the creation of indexes on a portion of a column (for example, on the first ten characters of a column), you can also build short indexes on very large columns.

MySQL can redefine the datatype of an existing column. To avoid losing any data, the values contained in the column must be compatible with the new datatype. For example, a date column could be redefined to a character datatype, but not a character datatype to an integer. Here's an example:

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

MySQL allows some additional flexibility in the *ALTER TABLE* statement by allowing users to issue multiple *ADD*, *ALTER*, *DROP*, and *CHANGE* clauses in a comma-delimited list in a single *ALTER TABLE* statement. However, be aware that the *CHANGE column_name* and *DROP INDEX* clauses are MySQL extensions not found in SQL2003. MySQL supports the clause *MODIFY column_name* to mimic the same feature found in Oracle.

Oracle

The Oracle syntax for *CREATE TABLE* creates a relational table either by declaring the structure or by referencing an existing table. You can modify a table after it is created using the *ALTER TABLE* statement. Oracle also allows the creation of a relational table that uses user-defined types for column definitions. You can also create an *object table* that is explicitly defined to hold a specific UDT, usually a *VARRAY* or *NESTED TABLE* type, as well as XMLType tables.

The standard ANSI style *CREATE TABLE* statement is supported, but Oracle has added many sophisticated extensions to the command. For example, Oracle allows significant control over the storage and performance parameters of a table. In both the *CREATE TABLE* and *ALTER TABLE* statements, you'll see a great deal of nesting and reusable clauses. To make this somewhat easier to read, we have broken Oracle's *CREATE TABLE* statement into three distinct variations (relational table, object table, and XML table) so that you can more easily follow the syntax.

The *CREATE TABLE* syntax for a standard relational table, which has no object or XML properties, is as follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
  [( {column | attribute} [SORT] [DEFAULT expression] [{column_constraint |
```

```
    inline_ref_constraint} ] |
    { table_constraint_clause | table_ref_constraint} |
    { GROUP log_group ( column [NO LOG] [,...] ) [ALWAYS] | DATA
      ( constraints [,...] ) COLUMNS } ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[table_constraint_clause]
{ [physical_attributes_clause] [TABLESPACE tablespace_name]
  [storage_clause] [[NO]LOGGING] |
  [CLUSTER ( column [,...] ) ] |
  {[ORGANIZATION
  {HEAP [physical_attributes_clause][TABLESPACE
    tablespace_name][storage_clause]
    [COMPRESS | NOCOMPRESS] [[NO]LOGGING] |
  INDEX [physical_attributes_clause][TABLESPACE
    tablespace_name][storage_clause]
    [PCTTHRESHOLD int] [COMPRESS [int] | NOCOMPRESS]
    [MAPPING TABLE | NOMAPPING][...] [[NO]LOGGING]
    [[INCLUDING column] OVERFLOW
    [physical_attributes_clause][TABLESPACE tablespace_name]
    [storage_clause] [[NO]LOGGING] } } ] |
  EXTERNAL ( [TYPE driver_type] ) DEFAULT DIRECTORY directory_name
    [ACCESS PARAMETERS {USING CLOB subquery | ( opaque_format ) } ]
    LOCATION ( [directory_name.:]'location_spec' [,...] )
    [REJECT LIMIT {int | UNLIMITED} } } }
[ {ENABLE | DISABLE} ROW MOVEMENT ]
[[NO]CACHE] [[NO]MONITORING] [[NO]ROWDEPENDENCIES]
[PARALLEL int | NOPARALLEL] [NOSORT] [[NO]LOGGING]
[COMPRESS [int] | NOCOMPRESS]
[ {ENABLE | DISABLE} [[NO]VALIDATE]
  {UNIQUE ( column [,...] ) | PRIMARY KEY | CONSTRAINT constraint_name } ]
  [USING INDEX {index_name | CREATE_INDEX_statement} ] [EXCEPTIONS INTO]
  [CASCADE] [ {KEEP | DROP} INDEX ] ] |
[partition_clause]
[AS subquery]
```

The relational table syntax contains a large number of optional clauses. However, the table definition must contain, at a minimum, either column names and datatypes specifications or the AS *subquery* clause.

The Oracle syntax for an object table follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
```

```
AS object_type [[NOT] SUBSTITUTABLE AT ALL LEVELS]
[ ( { column | attribute } [DEFAULT expression] [{ column_constraint |
    inline_ref_constraint } ] |
    { table_constraint_clause | table_ref_constraint } |
    { GROUP log_group (column [NO LOG] [,...]) [ALWAYS] | DATA
    (constraints [,...]) COLUMNS } ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX [index_name] ([physical_attributes_clause] [storage_clause]) ]
[physical_attributes_clause] [TABLESPACE tablespace_name] [storage_clause]
```

The Oracle syntax for an XMLType table follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
OF XMLTYPE
[ ( { column | attribute } [DEFAULT expression] [{ column_constraint |
    inline_ref_constraint } ] |
    { table_constraint_clause | table_ref_constraint } |
    { GROUP log_group (column [NO LOG] [,...]) [ALWAYS] | DATA
    (constraints [,...]) COLUMNS } ) ]
[XMLTYPE {OBJECT RELATIONAL [xml_storage_clause] |
    CLOB [lob_segname] [lob_params]}] [xml_schema_spec]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX index_name ([physical_attributes_clause] [storage_clause]) ]
[physical_attributes_clause] [TABLESPACE tablespace_name] [storage_clause]
```

The Oracle syntax for *ALTER TABLE* changes the table or column properties, storage characteristics, *LOB* or *VARRAY* properties, partitioning characteristics, and integrity constraints associated with a table and/or its columns. The statement does other things like move an existing table into a different tablespace, free recuperated space, compact the table segment, and adjust the "high-water mark."

The ANSI SQL standard uses the *ALTER* keyword to modify existing elements of a table, while Oracle uses *MODIFY* to mean the same thing. Since they are essentially the same thing, please consider the behavior of otherwise identical clauses (for example, ANSI's *ALTER TABLE...ALTER COLUMN* to Oracle's *ALTER TABLE...MODIFY COLUMN*) to be functionally equivalent.

Oracle's *ALTER TABLE* syntax is:

```
ALTER TABLE table_name
-- alter table characteristics
[physical_attributes_clause] [storage_clause]
[[NO]LOGGING] [[NO]CACHE] [[NO]MONITORING] [[NO]COMPRESS]
[SHRINK SPACE [COMPACT] [CASCADE] ]
[UPGRADE [[NOT] INCLUDING DATA] column_name datatype attributes]
```

```
[[NO]MINIMIZE RECORDS_PER_BLOCK]

[PARALLEL int | NOPARALLEL]

[{ENABLE | DISABLE} ROW MOVEMENT]

[{ADD | DROP} SUPPLEMENTAL LOG

  {GROUP log_group [(column_name [NO LOG] [...]) [ALWAYS] ] |
  DATA ( {ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY} [...] ) COLUMNS} ]

[ALLOCATE EXTENT

  [( SIZE int[K|M|G|T] ) [DATATILE 'filename' ] [INSTANCE int] ) ] ]

[DEALLOCATE UNUSED [KEEP int[K|M|G|T] ] ]

[ORGANIZATION INDEX ...

  [COALESCE] [MAPPING TABLE | NOMAPPING] [{PCTTHRESHOLD int]

  [COMPRESS int | NOCOMPRESS]

  [ { ADD OVERFLOW [TABLESPACE tablespace_name] [[NO]LOGGING

    [physical_attributes_clause] ] |

  OVERFLOW { [ALLOCATE EXTENT (SIZE int [K | M]) [DATAFILE

    'filename' ] [INSTANCE int] ) |

    [DEALLOCATE UNUSED [KEEP int [K | M]] } } ] ]

[RENAME TO new_table_name]

-- alter column characteristics

[ADD (column_name datatype attributes [...] ) ]

[DROP { {UNUSED COLUMNS | COLUMNS CONTINUE} [CHECKPOINT int] |

  {COLUMN column_name | (column_name [...]) } [CHECKPOINT int]

  [{CASCADE CONSTRAINTS | INVALIDATE}] } ]

[SET UNUSED {COLUMN column_name | (column_name [...]) }

  [{CASCADE CONSTRAINTS | INVALIDATE}]}

[MODIFY { (column_name datatype attributes [...] ) |

  COLUMN column_name [NOT] SUBSTITUTABLE AT ALL LEVELS [FORCE] } ]

[RENAME COLUMN old_column_name TO new_column_name]

[MODIFY {NESTED TABLE | VARRAY} collection_item [RETURN AS {LOCATOR |

  VALUE}]]

-- alter constraint characteristics

[ADD CONSTRAINT constraint_name table_constraint_clause]

[MODIFY CONSTRAINT constraint_name constraint_state_clause]

[RENAME CONSTRAINT old_constraint_name TO new_constraint_name]

[DROP { { PRIMARY KEY | UNIQUE (column [...]) } [CASCADE]

  [{KEEP | DROP} INDEX] |

  CONSTRAINT constraint_name [CASCADE] } ]

-- alter table partition characteristics

[alter partition clauses]
```

```
-- alter external table characteristics

DEFAULT DIRECTORY directory_name

    [ACCESS PARAMETERS {USING CLOB subquery | ( opaque_format ) } ]

    LOCATION ( [ directory_name: ]'location_spec' [,...] )

[ADD ( column_name ... )][DROP column_name ...][MODIFY ( column_name ... )]

[PARALLEL int | NOPARALLEL]

[REJECT LIMIT { int | UNLIMITED } ]

[PROJECT COLUMN {ALL | REFERENCED} ] } }

-- move table clauses

[MOVE [ONLINE] [ physical_attributes_clause ]

    [TABLESPACE tablespace_name] [[NO]LOGGING] [PCTTHRESHOLD int]

    [COMPRESS int | NOCOMPRESS] [MAPPING TABLE | NOMAPPING]

    [[INCLUDING column] OVERFLOW

    [ physical_attributes_clause ] [TABLESPACE tablespace_name]

    [[NO]LOGGING]] ]

[LOB ...] [VARRAY ...] [PARALLEL int | NOPARALLEL ]

-- enable/disable attributes and constraints

[ { {ENABLE | DISABLE} [[NO]VALIDATE]} {UNIQUE ( column [,...] ) |

    PRIMARY KEY | CONSTRAINT constraint_name } ]

[USING INDEX { index_name | CREATE_INDEX_statement |

    [TABLESPACE tablespace_name] [ physical_attributes_clause ]

    [storage_clause]

    [NOSORT] [[NO]LOGGING] [ONLINE] [COMPUTE STATISTICS]

    [COMPRESS | NOCOMPRESS] [REVERSE]

    [ {LOCAL | GLOBAL} partition_clause ]

    [EXCEPTIONS INTO table_name] [CASCADE] [ {KEEP | DROP} INDEX ] ] |

    [ {ENABLE | DISABLE} ] [ {TABLE LOCK | ALL TRIGGERS} } ] }
```

The parameters are as follows:

column_constraint

Specifies a column constraint using the syntax found in the description later.

GROUP *log_group* (*column* [NO LOG] [,...]) [ALWAYS] | DATA (*constraints* [,...]) COLUMNS

Specifies a log group rather than a single logfile for the table.

ON COMMIT {DELETE | PRESERVE} ROWS

Declares whether a declared temporary table should keep the data in the table active for the entire session (PRESERVE) or only for the transaction that the temporary table is created in (DELETE).

table_constraint_clause

Specifies a table constraint using the syntax found in the description later.

physical_attributes_clause

Specifies the physical attributes of the table using the syntax found in the description later.

TABLESPACE *tablespace_name*

Specifies the name of the tablespace where the table you are creating will be stored. If omitted, the default tablespace for the schema owner will be used. See below for specifics. See the Oracle Concepts manual to learn about tablespaces and their use.

storage_clause

Specifies physical storage characteristics of the table using the syntax found in the description later.

[NO]LOGGING

Specifies that redo log records will be written during object creation (*LOGGING*) or not (*NOLOGGING*). *LOGGING* is the default. In case of database failure under the *NOLOGGING* option, the operation cannot be recovered by applying logfiles, and the object must be recreated. *NOLOGGING* can speed the creation of database objects. The *LOGGING* clause replaces the older *RECOVERABLE* clause, which is deprecated.

CLUSTER (*column* [,...])

Declares that the table is part of a clustered index. The *column* list should correspond, one to one, with the columns in a previously declared clustered index. Because it uses the clustered indexes space allocation, the *CLUSTER* clause is compatible with the *physical_attributes_clause*, *storage_clause*, or *TABLESPACE* clause. Tables containing *LOBs* are incompatible with the *CLUSTER* clause.

ORGANIZATION HEAP

Declares how the data of the table should be recorded to disk. *HEAP*, the default for an Oracle table, declares that no order is associated with the storage of rows of data (i.e., the physical order in which records are written to disk) for this table. The *ORGANIZATION HEAP* clause allows several optional clauses, described in detail elsewhere in this list, control storage, logging, and compression for the table.

ORGANIZATION INDEX

Declares how the data of the table should be recorded to disk. *INDEX* declares that the records of the table are physically written to disk in the sort order defined by the primary key of the table. Oracle calls this an index-organized table. A primary key is required. Note that the *physical_attributes_clause*, the *TABLESPACE* clause, and the *storage_clause* — all described in greater detail elsewhere in this section — as well as the *[NO]LOGGING* keyword may all be associated with the new *INDEX* segment as you create it. In addition, the following subclauses may also be associated with an *ORGANIZATION INDEX* clause:

PCTTHRESHOLD *int*

Declares the percentage, *int*, of space in each index block preserved for data. Data, on a record by record basis, that cannot fit in this space will be placed in the overflow segment.

INCLUDING *column*

Declares the point at which a record is to split between index and overflow portions. All columns that follow the specified *column* will be stored in the overflow segment. The *column* cannot be a primary key column.

MAPPING TABLE | NOMAPPING

Tells the database to create a mapping of local to physical ROWIDs. This mapping is required to create a bitmap index on an IOT. Mappings are also partitioned identically if the table is partitioned. *NOMAPPING* tells the database not to create the ROWID map.

[INCLUDING *column*] OVERFLOW

Declares that a record that exceeds the *PCTTHRESHOLD* value be placed in a segment described in this clause. The *physical_attributes_clause*, the *TABLESPACE* clause, the *storage_clause* — all described elsewhere in the list in greater detail — and the *[NO]LOGGING* keyword may all be associated with a specific *OVERFLOW* segment when you create it. The optional *INCLUDING column* clause defines a column at which to divide an IOT row into index and overflow portions. Primary key columns are always stored in the index. However, all non-primary key columns that follow *column* are stored in the overflow data segment.

ORGANIZATION EXTERNAL

Declares how the data of the table should be recorded to disk. *EXTERNAL* declares that the table stores its data outside of the database and is usually read-only. Its metadata is stored in the database but its data is stored outside of the database. There are some restrictions on external tables. They cannot be temporary. They cannot have constraints. They can only have column, datatype, and attribute column-descriptors. *LOB* and *LONG* datatypes are disallowed. No other *ORGANIZATION* clauses are allowed with *EXTERNAL*. The following subclauses may be used with the *ORGANIZATION EXTERNAL* clause:

TYPE *driver_type*

Defines the access driver API for the external table. The default is *ORACLE_LOADER*.

DEFAULT DIRECTORY *directory_name*

Defines the default directory on the filesystem where the external table resides.

ACCESS PARAMETERS {USING CLOB *subquery* | (*opaque_format*) }

Assigns and passes specific parameters to the access driver. Oracle does not interpret this information. *USING CLOB subquery* tells Oracle to derive the parameters and their values from a subquery that returns a single row with a single column of datatype *CLOB*. The subquery cannot contain an *ORDER BY*, *UNION*, *INTERSECT*, or *MINUS/EXCEPT*. The *opaque_format* clause allows you to list parameters and their values as described in the ORACLE LOADER section of the "Oracle9i Database Utilities" guide.

LOCATION (*directory_name*:'*location_spec*' [,...])

Defines one or more external data sources, usually as files. Oracle does not interpret this information.

REJECT LIMIT {*int* | UNLIMITED}

Defines the number of conversion errors, *int*, that are allowed during the query to the external data source before Oracle aborts the query and returns an error. *UNLIMITED* tells Oracle to continue with the query no matter how many errors are encountered. The default is 0.

{*ENABLE* | *DISABLE*} ROW MOVEMENT

Specifies that a row may be moved to a different partition or subpartition if required due to an update of the key (*ENABLE*) or not (*DISABLE*). The *DISABLE* keyword also specifies that Oracle return an error if an update to a key would require a move.

[NO]CACHE

Buffers a table for rapid reads, while *NOCACHE* turns off this behavior. Index-organized tables offer *CACHE* behavior.

[NO]MONITORING

Specifies that modification statistics can be collected for this table (*MONITORING*) or not (*NOMONITORING*). *NOMONITORING* is the default.

[NO]ROWDEPENDENCIES

Specifies whether a table will use *row-level dependency tracking*, a feature that applies a system change number (SCN) greater than or equal to the last time of the transaction affecting the row. The SCN adds 6 extra bytes of space to each record. Row-level dependency tracking is most useful in replicated environments with parallel data propagation. *NOROWDEPENDENCIES* is the default.

PARALLEL *int* | NOPARALLEL

Allows for the parallel creation of the table by distinct CPUs to speed the operation. It also enables parallelism for queries and other data-manipulation operations against the table after its creation. An optional integer value may be supplied to define the exact number of parallel threads used to create the table in parallel, as well as the parallel threads allowed to service the table in the future. (Oracle calculates the best number of threads to use in a given parallel operation, so the *PARALLEL* clause is an optional feature.) *NOPARALLEL*, the default, creates the table serially and disallows future parallel queries and data-manipulation operations.

COMPRESS [*int*] | NOCOMPRESS

Specifies if the table should be compressed or not. On index-organized tables, only the key is compressed; on heap-organized tables, the entire table is compressed. This can greatly reduce the amount of space consumed by the table. *NOCOMPRESS* is the default. In index-organized tables, you can specify the number of prefix columns, *int*, to compress. The default value for *int* is the number of keys in the primary key minus one. You need not specify an *int* value on other clauses such as *ORGANIZATION*. When you omit the *int* value, Oracle will apply compression to the entire table.

{ENABLE | DISABLE} [[NO]VALIDATE]] {UNIQUE (*column* [,...]) | PRIMARY KEY | CONSTRAINT *constraint_name*}

Declares whether the named key or constraint applies to all of the data in the new table. *ENABLE* specifies that the key or constraint applies to all new data in the table. *DISABLE* specifies that the key or constraint is disabled for the new table, with the following options:

[NO]VALIDATE

Verifies that all existing data in the table complies with the key or constraint (*VALIDATE*) or not (*NOVALIDATE*). When *NOVALIDATE* is specified with *ENABLE*, Oracle does not verify that existing data in the table complies with the key or constraint, but ensures that new data added to the table does comply with the constraint.

{UNIQUE (*column* [,...]) | PRIMARY KEY | CONSTRAINT *constraint_name*}

Declares the unique constraint, primary key, or constraint that is enabled or disabled.

USING INDEX *index_name* | *CREATE_INDEX_statement*

Declares the name, *index_name*, of a pre-existing index (and its characteristics) used to enforce the key or constraint, or creates a new index, *CREATE_INDEX_statement*. If neither are declared, Oracle creates a new index.

EXCEPTIONS INTO

Specifies the name of a table into which Oracle places information about rows violating the constraint. Run the *utlexpt1.sql* script before using this keyword to explicitly create this table.

CASCADE

Cascades the disablement/enablement to any integrity constraints that depend on the constraint named in the clause. Usable only with the *DISABLE* clause.

{KEEP | DROP} INDEX

Lets you keep (*KEEP*) or drop (*DROP*) an index used to enforce a unique or primary key. You can drop the key only when disabling it.

partition_clause

Declares partitioning and subpartitioning of a table. Partitioning syntax can be quite complex. Refer to the material later in this section under [Oracle partitioned and subpartitioned tables](#) for the full syntax and examples.

AS subquery

Declares a *subquery* that inserts rows into the table upon creation. The column names and datatypes used in the subquery can act as substitutes for column name and attribute declarations for the table.

AS object_type

Declares that the table is based on a pre-existing object type.

[NOT] SUBSTITUTABLE AT ALL LEVELS

Declares whether row objects corresponding to subtypes can be inserted into the type table or not. When this clause is omitted, the default is *SUBSTITUTABLE AT ALL LEVELS*.

inline_ref_constraint and *table_ref_constraint*

Declares a reference constraint used by an object-type table or XMLType table. These clauses are described in greater detail later in this section under [Oracle XMLType and object-type tables](#).

OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}

Declares whether the object ID (OID) of the object-type table is *SYSTEM GENERATED* or based on the *PRIMARY KEY*. When omitted, the default is *SYSTEM GENERATED*.

OIDINDEX [index_name]

Declares an index, and possibly a name for the index, if the OID is system generated. You may optionally apply a *physical_attributes_clause* and a *storage_clause* to the *OIDINDEX*. If the OID is based on the primary key, then this clause is unnecessary.

OF XMLTYPE

Declares that the table is based on Oracle's *XMLTYPE* datatype.

XMLTYPE {OBJECT RELATIONAL [xml_storage_clause] | CLOB [lob_segname] [lob_params]}

Declares how the underlying storage of the *XMLTYPE* is handled. *OBJECT RELATIONAL* stores the data in object-relational columns. This subclause requires an *xml_schema_spec* and a schema that has been preregistered using the *DBMS_XMLSCHEMA* package. *CLOB* specifies that the *XMLTYPE* data will be stored in a *LOB* column. You may optionally specify the *LOB* segment name and/or the *LOB* storage parameters.

xml_schema_spec

Allows you to specify the URL of a registered XML schema and an XML element name. The element name is required but the URL is optional.

ADD ...

Adds a new column, constraint, overflow segment, or supplemental log group to an existing table.

MODIFY ...

Changes an existing column, constraint, or supplemental log group on an existing table.

DROP ...

Drops an existing column, constraint, or supplemental log group from an existing table. You can explicitly drop columns marked as unused from a table with *DROP UNUSED COLUMNS*; however, Oracle will also drop all unused columns when *any* other column is dropped. The *INVALIDATE* keyword causes any object that depends on the dropped object to become invalid and unusable until it is recompiled or when it is next used. The *COLUMNS CONTINUE* clause is used only when a *DROP COLUMN* statement failed with an error and you wish to continue where it left off.

RENAME ...

Renames an existing table, column, or constraint on an existing table.

SET UNUSED ...

Declares a column or columns to be unused. The column or columns are no longer accessible from *SELECT* statements, though they still count toward the maximum number of columns allowed per table (1000). *SET UNUSED* is the fastest way to render a column unusable within a table, but it is not the best way. Only use *SET UNUSED* as a shortcut until you can actually use *ALTER TABLE...DROP* to drop the column.

COALESCE

Merges the contents of index blocks used to maintain the index-organized table so that the blocks can be reused. *COALESCE* is similar to *SHRINK*, though *COALESCE* compacts the segments less densely than does *SHRINK* and it does not release unused space.

ALLOCATE EXTENT

Explicitly allocates a new extent for the table using the specified parameters. You may mix and match any of the parameters. The size of the extents may be in bytes (no suffix), kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T).

DEALLOCATE UNUSED [KEEP int[K | M | G | T]]

Deallocates unused space at the end of the table, *LOB* segment, partition, or subpartition. The deallocated space is then usable by other objects in the database. The *KEEP* keyword tells how much space you want to have left over after deallocation is complete.

SHRINK SPACE [COMPACT] [CASCADE]

Shrinks the table, index-organized table, index, partition, subpartition, materialized view, or materialized log view, though only segments in tablespaces with automatic segment management may be shrunk. Shrinking a segment moves rows in the table, so make sure *ENABLE ROW MOVEMENT* is also used in the *ALTER TABLE...SHRINK* statement. Oracle compacts the segment, releases the emptied space, and adjusts the high-water mark unless the optional keywords *COMPACT* and/or *CASCADE* are applied. The *COMPACT* keyword only defragments the segment space and compacts the table row for subsequent release, but it does not readjust the high-water mark nor empty the space immediately. The *CASCADE* keyword performs the same shrinking operation (with some restrictions and exceptions) on all dependent objects of the table, including secondary indexes on index-organized tables. Used only with *ALTER TABLE*.

UPGRADE [NOT] INCLUDING DATA

Converts the metadata of object tables and relational tables with object columns to the latest version for each referenced type. The *INCLUDING DATA* clause will convert the data to the latest type format (*INCLUDING DATA*) or leave it unchanged (*NOT INCLUDING DATA*).

MOVE ...

Moves the tablespace, index-organized table, partition, or subpartition to a new location on the filesystem.

[NO]MINIMIZE RECORDS_PER_BLOCK

Tells Oracle to restrict or leave open the number of records allowed per block. The *MINIMIZE* keyword tells Oracle to calculate the maximum number of records per block and set the limit at that number. (Best to do this when a representative amount of data is already in the table.) This clause is incompatible with nested tables and index-organized tables. *NOMINIMIZE* is the default.

PROJECT COLUMN {REFERENCE | ALL}

Determines how the driver for the external data source validates the rows of the external table in subsequent queries. *REFERENCE* processes only those columns in the select item list. *ALL* processes the values in all columns, even those not in the select item list, and validates rows with full and valid column entries. Under *ALL*, rows are rejected when errors occur, even on columns that are not selected. *ALL* returns consistent results, while *REFERENCE* returns varying numbers of rows depending on the columns referenced.

{ENABLE | DISABLE} {TABLE LOCK | ALL TRIGGERS}

Enables or disables table-level locks and all triggers on the table, respectively. *ENABLE TABLE LOCK* is required if you wish to change the structure against an existing table, but it is not required when changing or reading the data of the table.

A global temporary table is available to all user sessions, but the data stored within a global temporary table is visible only to the session that inserted it. The *ON COMMIT* clause, which is allowed only when creating temporary tables, tells Oracle to either truncate the table after each commit against the table (*DELETE ROWS*) or to truncate the table when the session terminates (*PRESERVE ROWS*). For example:

```
CREATE GLOBAL TEMPORARY TABLE shipping_schedule
(
  ship_date DATE,
  receipt_date DATE,
  received_by VARCHAR2(30),
  amt NUMBER)
ON COMMIT PRESERVE ROWS;
```

The *CREATE TABLE* statement shown above creates a global temporary table, **shipping_schedule**, which retains inserted rows across multiple transactions.

The Oracle *physical_attributes_clause*

The *physical_attributes_clause* (as shown in the following code block) defines storage characteristics for an entire local table, or if the table is partitioned, for a specific partition (discussed later). To declare the physical attributes of a new table or change the attributes on an existing table, simply declare the new values:

```
-- physical_attributes_clause
[{PCTFREE int | PCTUSED int | INITRANS int |
  MAXTRANS int | storage_clause}]
```

where:

PCTFREE int

Defines the percentage of free space reserved for each data block in the table. For example, a value of 10 reserves 10% of the data space for new rows to be inserted.

PCTUSED int

Defines the minimum percentage of space allowed in a block before it can receive new rows. For example, a value of 90 means new rows are inserted in the data block when the space used falls below 90%. The sum of *PCTFREE* and *PCTUSED* cannot exceed 100.

INITRANS int

Rarely tinkered with; it defines the allocation of from 1 to 255 initial transactions to a data block.

MAXTRANS int

Defines the maximum number of concurrent transactions on a data block.

The Oracle storage_clause and LOBs

The *storage_clause* controls a number of attributes governing the physical storage of data:

-- *storage_clause*

```
STORAGE ( [ {INITIAL int [K | M]
          | NEXT int [K | M]
          | MINEXTENTS int
          | MAXEXTENTS { int | UNLIMITED}
          | PCTINCREASE int
          | FREELISTS int
          | FREELIST GROUPS int
          | BUFFER_POOL {KEEP | RECYCLE | DEFAULT} ] [...] )
```

When delineating the storage clause attributes, enclose them in parentheses and separate them with spaces—for example, (*INITIAL 32M NEXT8M*). The attributes are as follows:

INITIAL int [K | M]

Sets the initial extent size of the table in bytes, kilobytes (K), or megabytes (M). Oracle 10g allows [K|M|G|T] to describe the size allocation in bytes, kilobytes (K), megabytes (M), gigabytes (G), and terabytes (T).

NEXT int [K | M]

Tells how much additional space to allocate after *INITIAL* is filled.

MINEXTENTS int

Tells Oracle to create a minimum number of extents. By default, only one is created, but more can be created when the object is initialized.

MAXEXTENTS int

Tells Oracle the maximum extents allowed. This value may be set to *UNLIMITED*. (Note that *UNLIMITED* should be used with caution since a table could grow until it consumes all free space on a disk.)

PCTINCREASE int

Controls the growth rate of the object after the first growth. The initial extent gets allocated as specified. The second extent is the size specified by *NEXT*. The third extent is *NEXT + (NEXT * PCTINCREASE)*. When *PCTINCREASE* is set to 0, *NEXT* is always used. Otherwise, each added extent of storage space is *PCTINCREASE* larger than the previous extent.

FREELISTS int

Establishes the number of freelists for each group, defaulting to 1.

FREELIST GROUPS int

Sets the number of groups of freelists, defaulting to 1.

BUFFER_POOL {KEEP | RECYCLE | DEFAULT}

Specifies a default buffer pool or cache for any non-cluster table where all object blocks are stored. Index-organized tables may have a separate buffer pool for the index and overflow segments. Partitioned tables inherit the buffer pool from the table definition unless they are specifically assigned a separate buffer pool.

KEEP

Puts object blocks into the *KEEP* buffer pool; that is, directly into memory. This enhances performance by reducing I/O operations on the table. *KEEP* takes precedence over the *NOCACHE* clause.

RECYCLE

Puts object blocks into the *RECYCLE* buffer pool.

DEFAULT

Puts object blocks into the *DEFAULT* buffer pool. When omitted, *DEFAULT* is the default buffer pool behavior.

For example, the table **books_sales** is defined on the **sales** tablespace as consuming an initial 8 MB of space, to grow by no less than 8 MB when the first extent is full. The table has no less than 1 and no more than 8 extents, limiting its maximum size to 64 MB:

```
CREATE TABLE book_sales
  (qty NUMBER,
   period_end_date DATE,
   period_nbr NUMBER)
TABLESPACE sales
STORAGE (INITIAL 8M NEXT 8M MINEXTENTS 1 MAXEXTENTS 8);
```

An example of a *LOB* table called **large_objects** with special handling for text and image storage might look like:

```
CREATE TABLE large_objects
  (pretty_picture BLOB,
   interesting_text CLOB)
STORAGE (INITIAL 256M NEXT 256M)
LOB (pretty_picture, interesting_text)
  STORE AS (TABLESPACE large_object_segment
            STORAGE (INITIAL 512M NEXT 512M)
            NOCACHE LOGGING);
```

The exact syntax used to define a *LOB*, *CLOB*, or *NLOB* column is defined by the *lob_parameter_clause*. *LOBs* can appear at many different levels within an Oracle table. For instance, separate *LOB* definitions could exist in a single table in a partition definition, a subpartition definition, and at the top table-level. The syntax of *lob_parameter_clause* follows:

```
{TABLESPACE tablespace_name} [{ENABLE | DISABLE} STORAGE IN ROW]
[storage_clause] [CHUNK int] [PCTVERSION int]
[RETENTION] [FREEPOOLS int]
```

[[CACHE | {NOCACHE | CACHE READS} [{LOGGING | NOLOGGING}]] }

In the *lob_parameter_clause*, each parameter is identical to those of the wider *CREATE TABLE LOB*-object level. However, the following four parameters are unique to *LOBs*:

{ENABLE | DISABLE} STORAGE IN ROW

Defines whether the *LOB* value is stored inline with the other columns of the row and the *LOB* locator (*ENABLE*), when it is smaller than approximately 4,000 bytes or less, or outside of the row (*DISABLE*). This setting cannot be changed once it is set.

CHUNK *int*

Allocates *int* number of bytes for *LOB* manipulation. *int* should be a multiple of the database blocksize, otherwise Oracle will round up. *int* should also be less than or equal to the value of *NEXT*, from the *storage_clause*, or else an error results. When omitted, the default chunk size is one block. This setting cannot be changed once it is set.

PCTVERSION *int*

Defines the maximum percentage, as *int*, of the overall *LOB* storage dedicated to maintaining old versions of the *LOB*. When omitted, the default is 10%.

RETENTION

Used in place of *PCTVERSION* on databases in automatic undo mode. *RETENTION* tells Oracle to retain old versions of the *LOB*.

The following example shows our **large_objects** *LOB* table with added parameters to control inline storage and retention of old *LOBs*:

```
CREATE TABLE large_objects
```

```
  (pretty_picture BLOB,
```

```
   interesting_text CLOB)
```

```
STORAGE (INITIAL 256M NEXT 256M)
```

```
LOB (pretty_picture, interesting_text)
```

```
  STORE AS (TABLESPACE large_object_segment
```

```
    STORAGE (INITIAL 512M NEXT 512M)
```

```
    NOCACHE LOGGING
```

```
    ENABLE STORAGE IN ROW
```

```
    RETENTION);
```

The earlier example added parameter values for *STORAGE IN ROW* and *RETENTION*. But since we did not set one for *CHUNK*, that value is set to the Oracle default for the *LOB*.

Oracle nested tables

Oracle allows the declaration of a *NESTED TABLE* in which a table is virtually stored within a column of another table. The *STORE AS* clause enables a proxy name for the table within a table, but the nested table must be created initially as a user-defined datatype. This capability is valuable for sparse arrays of values, but we don't recommend it for day-to-day tasks. For example:


```
CREATE TYPE prop_nested_tbl AS TABLE OF props_nt;  
  
CREATE TABLE proposal_types  
  (proposal_category VARCHAR2(50),  
   proposals          PROPS_NT)  
  
NESTED TABLE props_nt STORE AS props_nt_table;
```

This example creates a table called **proposal_types** along with a nested table called **props_nt**, which is stored as **props_nt_table**.

Oracle compressed tables

Oracle 10g (and Oracle 9i Release 2) allows compression of both keys and entire tables. (Oracle 9i allowed only key compression.) Although compression adds a tiny bit of overhead, it significantly reduces that amount of disk space consumed by a table. This is especially useful for databases pushing the envelope in terms of size. Key compression is handled in the *ORGANIZE INDEX* clause, while table compression is handled in the *ORGANIZE HEAP* clause.

Oracle partitioned and subpartitioned tables

Oracle allows tables to be partitioned and subpartitioned. You can also break out *LOBs* onto their own partition(s). A partitioned table may be broken into distinct parts, possibly placed on separate disk subsystems to improve I/O performance, based on four strategies: range, hash, list, and a composite of the first three. Partitioning syntax is quite elaborate:

```
{ PARTITION BY RANGE (column [...]) (PARTITION [partition_name]  
  VALUES LESS THAN ({MAXVALUE | value} [...])  
  [table_partition_description] ) |  
PARTITION BY HASH (column [...])  
  {(PARTITION [partition_name] [partition_storage_clause] [...]) |  
  PARTITIONS hash_partition_qty [STORE IN (tablespace [...])]  
  [OVERFLOW STORE IN (tablespace [...])]} |  
PARTITION BY LIST (column [...]) (PARTITION [partition_name]  
  VALUES ({MAXVALUE | value} [...])  
  [table_partition_description] |  
PARTITION BY RANGE (column [...])  
  {subpartition_by_list | subpartition_by_hash}  
  (PARTITION [partition_name] VALUES LESS THAN  
    ({MAXVALUE | value} [...])  
    [table_partition_description] ) }
```

The following example code shows the **orders** table partitioned by range:

```
CREATE TABLE orders  
  (order_number NUMBER,  
   order_date   DATE,  
   cust_nbr     NUMBER,  
   price        NUMBER,  
   qty          NUMBER,
```

```
    cust_shp_id NUMBER)
PARTITION BY RANGE(order_date)
(PARTITION pre_yr_2000 VALUES LESS THAN
    TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
PARTITION pre_yr_2004 VALUES LESS THAN
    TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
PARTITION post_yr_2004 VALUES LESS THAN
    MAXVALUE) );
```

This example creates three partitions on the **orders** table — one for the orders taken before the year 2000 (**pre_yr_2000**), one for the orders taken before the year 2004 (**pre_yr_2004**), and another for the orders taken after the year 2004 (**post_yr_2004**) — all based on the range of dates that appear in the **order_date** column.

The next example creates the **orders** table based on a hash value in the **customer_shipping_id**:

```
CREATE TABLE orders
(
    order_number NUMBER,
    order_date DATE,
    cust_nbr NUMBER,
    price NUMBER,
    qty NUMBER,
    cust_shp_id NUMBER)
PARTITION BY HASH (cust_shp_id)
(PARTITION shp_id1 TABLESPACE tblspc01,
PARTITION shp_id2 TABLESPACE tblspc02,
PARTITION shp_id3 TABLESPACE tblspc03)
ENABLE ROW MOVEMENT;
```

The big difference in how the records are divided among partitions between the hash partition example and the range partition example is that the range partition code explicitly defines where each record goes, while the hash partition example allows Oracle to decide (by applying a hash algorithm) in which partition to place the record. (Note that we also enabled row movement for the table.)

Examples of list and composite partitioning are shown with the subpartitioning information detailed in the next paragraph. In addition to breaking tables apart into partitions (for easier backup, recovery, or performance reasons), you may further break it apart into subpartitions. The *subpartition_by_list* clause syntax follows:

```
SUBPARTITION BY LIST (column)
[SUBPARTITION TEMPLATE
{ (SUBPARTITION subpartition_name
    [VALUES {DEFAULT | {val | NULL} [,...]}
    [partitioning_storage_clause] ) |
    hash_subpartition_qty } ]
```

For example, we'll recreate the **orders** table once again, this time using a range-hash composite partition. In a range-

hash composite partition, the partitions are broken apart by range values, while the subpartitions are broken apart by hashed values. List partitions and subpartitions are broken apart by a short list of specific values. Because you must list out all the values by which the table is partitioned, the partition value is best taken from a small list of values. In this example, we've added a column (**shp_region_id**) that allows four possible regions:

```
CREATE TABLE orders
(
  order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20) )
PARTITION BY RANGE(order_date)
SUBPARTITION BY LIST(shp_region)
SUBPARTITION TEMPLATE(
  (SUBPARTITION shp_region_north
    VALUES ('north','northeast','northwest'),
  SUBPARTITION shp_region_south
    VALUES ('south','southeast','southwest'),
  SUBPARTITION shp_region_central
    VALUES ('midwest'),
  SUBPARTITION shp_region_other
    VALUES ('alaska','hawaii','canada')
(PARTITION pre_yr_2000 VALUES LESS THAN
  TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
PARTITION pre_yr_2004 VALUES LESS THAN
  TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
PARTITION post_yr_2004 VALUES LESS THAN
  MAXVALUE) )
ENABLE ROW MOVEMENT;
```

This code example not only sends the records of the table to one of three partitions based on the **order_date**, it further partitions the records into one of four subpartitions based on the region where the order is being shipped and on the value of the **shp_region** column. By using the *SUBPARTITION TEMPLATE*, you apply the same set of subpartitions to each partition. You can manually override this behavior by specifying subpartitions for each partition.

You may also subpartition using a hashing algorithm. The *subpartition_by_hash* clause follows:

```
SUBPARTITION BY HASH (column [,...])
{SUBPARTITIONS qty [STORE IN (tablespace_name [,...]) ] |
SUBPARTITION TEMPLATE
{ (SUBPARTITION subpartition_name [VALUES {DEFAULT | {val | NULL}
[,...]} [partitioning_storage_clause] ) |
hash_subpartition_qty }
```

The *table_partition_description* clause referenced in the partitioning syntax (in the earlier section [Oracle partitioned and subpartitioned tables](#)) is, in itself, very flexible and supports precise handling of *LOB* and *VARRAY* data:

```
[segment_attr_clause] [[NO] COMPRESS [int]] [OVERFLOW segment_attr_clause]
[partition_level_subpartition_clause]
[ LOB { (lob_item [,...]) STORE AS lob_param_clause |
  (lob_item) STORE AS {lob_segname (log_param_clause) |
  log_segname | (log_param_clause) } } |
VARRAY varray_item {{{[ELEMENT] IS OF [TYPE] (ONLY type_name) |
  [NOT] SUBSTITUTABLE AT ALL LEVELS}} STORE AS LOB { log_segname |
  [log_segname] (log_param_clause) } |
{{{[ELEMENT] IS OF [TYPE] (ONLY type_name) |
  [NOT] SUBSTITUTABLE AT ALL LEVELS}}}
```

The *partition_level_subpartition_clause* syntax follows:

```
{SUBPARTITIONS hash_subpartition_qty [STORE IN (tablespace_name [,...])] |
  SUBPARTITION subpartition_name [VALUES {DEFAULT | {val | NULL} [,...]}
  [partitioning_storage_clause] }
```

The *partition_storage_clause* (see the earlier section [Oracle partitioned and subpartitioned tables](#)), like the table-level *storage_clause* defined earlier, defines how elements of a partition (or subpartition) are stored. The syntax follows:

```
{{TABLESPACE tablespace_name} | [OVERFLOW TABLESPACE tablespace_name] |
VARRAY varray_item STORE AS LOB log_segname |
LOB (lob_item) STORE AS { (TABLESPACE tablespace_name) |
  Log_segname [(TABLESPACE tablespace_name)] }
```

In this final partitioning example, we'll again recreate the **orders** table using a composite range-hash partition, this time with *LOB* (actually, an *NLOB* column) and storage elements:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NLOB )
PARTITION BY RANGE(order_date)
```

```
SUBPARTITION BY HASH(cust_shp_id)
(PARTITION pre_yr_2000 VALUES LESS THAN
  TO_DATE('01-JAN-2000', 'DD-MON-YYYY') TABLESPACE tblspc01
  LOB (order_desc) STORE AS (TABLESPACE tblspc_a01
    STORAGE (INITIAL 10M NEXT 20M) )
  SUBPARTITIONS subpartn_a,
PARTITION pre_yr_2004 VALUES LESS THAN
  TO_DATE('01-JAN-2004', 'DD-MON-YYYY') TABLESPACE tblspc02
  LOB (order_desc) STORE AS (TABLESPACE tblspc_a02
    STORAGE (INITIAL 25M NEXT 50M) )
  SUBPARTITIONS subpartn_b TABLESPACE tblspc_x07,
PARTITION post_yr_2004 VALUES LESS THAN
  MAXVALUE (SUBPARTITION subpartn_1,
    SUBPARTITION subpartn_2,
    SUBPARTITION subpartn_3
    SUBPARTITION subpartn_4) )
ENABLE ROW MOVEMENT;
```

In this somewhat more complex example, we now define the **orders** table with the added *NLOB* table called **order_desc**. In the **pre_yr_2000** and **pre_yr_2004** partitions, we specify that all of the non-*LOB* data goes to tablespaces **tblspc01** and **tblspc02**, respectively. However, the *NLOB* values of the **order_desc** column will be stored in the **tblspc_a01** and **tblspc_a02**, respectively, with their own unique storage characteristics. Note that the subpartition **subpartn_b** under partition **pre_yr_2004** is also stored in its own tablespace, **tblspc_x07**. Finally, the last partition (**post_yr_2004**) and its subpartitions are stored in the default tablespace for the **orders** table because no partition- or subpartition-level *TABLESPACE* clause overrides the default.

Altering partitioned and subpartition tables

Anything about partitions and subpartitions that is explicitly set by the *CREATE TABLE* statement may be altered after the table is created. Many of the clauses shown later are merely repetitions of clauses that are described in the earlier section about creating partitioned tables; for example, the *SUBPARTITION TEMPLATE* clause or the *MAPPING TABLE* clause. Consequently, these clauses will not be repeated. Altering the partitions and/or subpartitions of an Oracle table is governed by this syntax:

```
ALTER TABLE table_name
[MODIFY DEFAULT ATTRIBUTES [FOR PARTITION partn_name]
  [physical_attributes_clause] [storage_clause] [PCTTHRESHOLD int]
  [{ADD OVERFLOW ... | OVERFLOW ...}] [[NO]COMPRESS]
  [{LOB (lob_name) | VARRAY varray_name} [(lob_parameter_clause)] ]
  [COMPRESS int | NOCOMPRESS]
[SET SUBPARTITION TEMPLATE {hash_subpartn_quantity |
  (SUBPARTITION subpartn_name [partn_list]
  [storage_clause])}]
[MODIFY PARTITION partn_name
  { [table_partition_description] |
```

```
[[REBUILD] UNUSABLE LOCAL INDEXES] |
[ADD [subpartn specification] ] |
[COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause] ] |
[{ADD | DROP} VALUES (partn_value [,...]) ] |
[MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSUED ...} ]
[MODIFY SUBPARTITION subpartn_name {hash_subpartn_attributes |
list_subpartn_attributes} ]
[MOVE {PARTITION | SUBPARTITION} partn_name
[MAPPING TABLE] [table_partition_description] [[NO]PARALLEL]
[update_index_clause] ]
[ADD PARTITION [partn_name] [p table_partition_description]
[[NO]PARALLEL] [update_index_clause] ]
[COALESCE PARTITION [[NO]PARALLEL] [update_index_clause] ]
[DROP {PARTITION | SUBPARTITION} partn_name [[NO]PARALLEL]
[update_index_clause] ]
[RENAME {PARTITION | SUBPARTITION} old_partn_name TO
```

new_partn_name]

```
[TRUNCATE {PARTITION | SUBPARTITION} partn_name
[{DROP | REUSE} STORAGE] [[NO]PARALLEL] [update_index_clause] ]
[SPLIT {PARTITION | SUBPARTITION} partn_name {AT | VALUES}
(value [,...])
[INTO (PARTITION [partn_name1]
[table_partition_description] ,
(PARTITION [partn_name2]
[table_partition_description] ) ) ]
[[NO]PARALLEL] [update_index_clause] ]
[MERGE {PARTITION | SUBPARTITION} partn_name1 , partn_name2
[INTO PARTITION [partn_name] [partn_attributes] ] [[NO]PARALLEL]
[update_index_clause] ]
[EXCHANGE {PARTITION | SUBPARTITION} partn_name WITH TABLE table_name
[{INCLUDING | EXCLUDING} INDEXES] [{WITH | WITHOUT} VALIDATION]
[[NO]PARALLEL] [update_index_clause] [EXCEPTIONS INTO table_name] ]
```

where:

MODIFY DEFAULT ATTRIBUTES [FOR PARTITION *partn_name*]

Modifies a wide variety of attributes for the current partition or a specific partition of *partn_name*. Refer to the earlier section [Oracle partitioned and subpartitioned tables](#) for all the details on partition attributes.

SET SUBPARTITION TEMPLATE { *hash_subpartn_quantity* | (SUBPARTITION *subpartn_name* [*partn_list*] [*storage_clause*]) }

Sets a new subpartition template for the table.

MODIFY PARTITION *partn_name*

Changes a wide variety of physical and storage characteristics, including the storage properties of *LOB* and *VARRAY* columns, of a pre-existing partition or subpartition called *partn_name*. Additional syntax may be appended to the *MODIFY PARTITION partn_name* clause:

```
{ [table_partition_description] | [[REBUILD] UNUSABLE LOCAL INDEXES] |  
  [ADD [subpartn_specification] ] |  
  [COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause]  
  { [{UPDATE | INVALIDATE} GLOBAL INDEXES] |  
    UPDATE INDEXES [ (index_name (  
      {index_partn | index_subpartn} ) ) [,...] ] } ] |  
  [{ADD | DROP} VALUES (partn_value [,...] ) ] |  
  [MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...}]
```

where:

table_partition_description

Described in the earlier section [Oracle partitioned and subpartitioned tables](#). This clause may be used on any partitioned table.

[REBUILD] UNUSABLE LOCAL INDEXES

Marks the local index partition as *UNUSABLE*. Adding the optional *REBUILD* keyword tells Oracle to rebuild the unusable local index partition as part of the operation performed by the *MODIFY PARTITION* statement. This clause may not be used with any other subclause of the *MODIFY PARTITION* statement nor may it be used on tables with subpartitions. This clause may be used on any partitioned table.

ADD [*subpartn_specification*]

Adds a hash or list subpartition specification, as described in the earlier section [Oracle partitioned and subpartitioned tables](#), to an existing range partition. This clause may be used to define range-hash or range-list composite partitions only. Oracle populates the new subpartition with rows from other subpartitions using either the hash function or the list values you specify. We recommend the total number of subpartitions be set to a power of 2 for optimal load balancing. You may add range-list subpartitions only if the table does not already have a *DEFAULT* subpartition. When adding range-list subpartitions, the list value clause is required but it cannot duplicate values found in any other subpartition of the current partition. The only storage or physical attribute you may couple with this clause for both range-hash and range-list subpartitions is the *TABLESPACE* clause.

COALESCE SUBPARTITION [[NO]PARALLEL] [*update_index_clause*]

Coalesces the subpartition of a range-hash composite partitioned table. This clause tells Oracle to distribute the contents of the last hash subpartitions to one or more remaining subpartition in the set, and then drop the last hash subpartition. Oracle also drops local index subpartitions corresponding to the subpartition you are coalescing. The *update_index_clause* is described later in this list. Global indexes may be updated or invalidated using the syntax {*UPDATE* | *INVALIDATE*} *GLOBAL INDEXES*. In addition, local indexes, index partitions, or index subpartitions may be updated using the syntax *UPDATE INDEXES* (*index_name* {*index_partn* | *index_subpartn*}).

{ADD | DROP} VALUES (*partn_value* [,...])

Adds a new value (or values) or drops existing values on an existing list-partitioned table, respectively. Local and global indexes are not affected by this clause. Values cannot be added to or dropped from a *DEFAULT* list partition.

MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSUED ...}

Defines a mapping table for a partitioned table that is an IOT. The *ALLOCATE EXTENT* and *DEALLOCATE UNUSUED* clauses are described earlier in the syntax description list of the "CREATE TABLE" statement. This clause may be used on any type of partitioned table, as long as the table is an index-organized table.

MODIFY SUBPARTITION *subpartn_name* {*hash_subpartn_attributes* | *list_subpartn_attributes*}

Modifies a specific hash or list subpartition according to the subpartition attributes described in the earlier section [Oracle partitioned and subpartitioned tables](#).

[MOVE {PARTITION | SUBPARTITION} *partn_name* [MAPPING TABLE] [*table_partition_description*] [[NO]PARALLEL] [*update_index_clause*]

Moves a specified partition (or subpartition) of *partn_name* to another partition (or subpartition) described in the *table_partition_description* clause, which is described in the earlier section [Oracle partitioned and subpartitioned tables](#). Moving a partition is I/O intensive, so the optional *PARALLEL* clause may be used to parallelize the operation. When omitted, *NOPARALLEL* is the default. In addition, you may optionally update or invalidate the local and global index as described in the *update_index_clause* shown below in this list.

ADD PARTITION [*partn_name*] [*table_partition_description*] [[NO]PARALLEL]

Adds a new partition (or subpartition) of *partn_name* to the table. The *ADD PARTITION* clause supports all aspects of creating a new partition or subpartition, via the *table_partition_description* clause, which is described in the earlier section [Oracle partitioned and subpartitioned tables](#). Adding a partition may be I/O intensive, so the optional *PARALLEL* clause may be used to parallelize the operation. When omitted, *NOPARALLEL* is the default. In addition, you may optionally update or invalidate local and global indexes on the table using the *update_index_clause* shown earlier in this list.

update_index_clause

Controls the status assigned to indexes once the partitions and/or subpartitions of a table are altered. By default, Oracle invalidates the entire index(es) of a table, not just those portions of the index on the partition and/or subpartition being altered. You may update or invalidate global indexes on the table *OR* update one or more specific index(es) on the table, respectively, using this syntax:

{ UPDATE | INVALIDATE } GLOBAL INDEXES]

UPDATE INDEXES ((*index_name* ({*index_partn*|*index_subpartn*})) [,...] }

COALESCE PARTITION [[NO]PARALLEL] [*update_index_clause*]

Takes the contents of the last partition of a set of hash partitions and reshapes the contents to one or more of the other partitions in the set. The last partition is then dropped. Obviously, this is only for use with hash partitions. The *update_index_clause*, shown earlier in this list, may be applied to update or invalidate the local and/or global indexes of the table being coalesced.

DROP {PARTITION | SUBPARTITION} *partn_name* [[NO]PARALLEL] [*update_index_clause*]

Drops an existing range or list partition or subpartition of *partn_name* from the table. The data within the partition is also dropped. If you want to keep the data, use the *MERGE PARTITION* clause. If you want to get rid of a hash partition or subpartition, use the *COALESCE PARTITION* clause instead. Tables with only a single partition are not affected by the *ALTER TABLE...DROP PARTITION* statement; instead, use the *DROP TABLE* statement.

RENAME {PARTITION | SUBPARTITION} *old_partn_name* TO *new_partn_name*

Renames an existing partition or subpartition of *old_partn_name* to a new name of *new_partn_name*.

TRUNCATE {PARTITION | SUBPARTITION} *partn_name* [{DROP | REUSE} STORAGE] [[NO]PARALLEL]

[*update_index_clause*]

Removes all of the rows of a partition or subpartition of *partn_name*. If you truncate a composition partition, all the rows of the subpartition(s) are also dropped. On IOTs, mapping table partitions and overflow partitions are also truncated. *LOB* data and index segments, if the table has any *LOB* columns, are also truncated. Finally, disable any existing referential integrity constraints on the data, or else delete the rows from the table first, then truncate the partition or subpartition. The optional *DROP STORAGE* and *REUSE STORAGE* subclauses define whether the space freed by the truncated data is made available for other objects in the tablespace or still allocated to the original partition or subpartition, respectively.

```
SPLIT {PARTITION | SUBPARTITION} partn_name {AT | VALUES} (value [...]) [INTO (PARTITION [partn_name1] [table_partition_description], (PARTITION [partn_name2] [table_partition_description]))] [ [NO]PARALLEL] [update_index_clause]
```

Creates from the current partition (or subpartition) of *partn_name* two new partitions (or subpartitions) called *partn_name1* and *partn_name2*. These new partitions may have their own complete specification, as defined by the *table_partition_description* clause. When omitted, the new partitions inherit all physical characteristics of the current partition. When splitting a *DEFAULT* partition, all of the split values go to *partn_name1* while all of the default values go to *partn_name2*. For IOTs, Oracle splits any mapping table partition in a manner corresponding to the split. Oracle also splits *LOB* and *OVERFLOW* segments, but you may also specify your own *LOB* and *OVERFLOW* storage characteristics, described in the earlier section [Oracle partitioned and subpartitioned tables](#). Additional syntax for this clause is described below:

```
{AT | VALUES} (value [...])
```

Splits range partitions (using *AT*) or list partitions (using *VALUES*) according to the *value* or *values* you specify. The *AT* (*value*) clause defines the new noninclusive upper range for the first of the two new partitions. The new value should be less than the partition bound of the current partition, but greater than the partition bound of the next lowest partition (if one exists). The *VALUES* (*value1*, *value2*, [...]) clause defines the values going into the first of the two new list partitions. The first new list partition is built from *value1* and the second is built from the remaining partition values in the current partition of *partn_name*. The value list must include values that already exist in the current partition, but it cannot contain all of the values of the current partition.

INTO

Defines the two new partitions that result from the split. At a minimum, the two *PARTITION* keywords, in parentheses, are required. Any characteristics not explicitly declared for the new partitions are inherited from the current partition of *partn_name*, including any subpartitioning. There are a few restrictions to note. When subpartitioning range-hash composition partitioned tables, only the *TABLESPACE* value is allowed for the subpartitions. Subpartitioning is not allowed at all when splitting range-list composite partitioned tables. Any indexes on heap-organized tables are invalidated by default when the table is split. You must use the *update_index_clause* to update their status.

```
[MERGE {PARTITION | SUBPARTITION} partn_name1, partn_name2 [INTO PARTITION [partn_name] [partn_attributes]] [ [NO]PARALLEL] [update_index_clause]
```

Merges the contents of two or more partitions or subpartitions of a table into a single new partition. Oracle then drops the two original partitions. Merged range partitions must be adjacent and are then bound by the higher boundary of the original two partitions when merged. Merged list partitions need not be adjacent and result in a single new partition with a union of the two sets of partition values. If one of the list partitions was the *DEFAULT* partition, then the new partition will be the default. Merged range-list composite partitions are allowed but may not have a new subpartition template. Oracle creates a subpartition template from the existing one(s) or, if none exist, creates a new *DEFAULT* subpartition. Physical attributes not defined explicitly are inherited from the table-level settings. By default, Oracle makes all local index partitions and global indexes *UNUSABLE* unless you override this behavior using the *update_index_clause*. (The exception to this rule is with IOTs, which, being index-based, will remain *USABLE* throughout the merge operation.) Merge operations are not allowed on hash-partitioned tables, so use the *COALESCE PARTITION* clause instead.

```
EXCHANGE {PARTITION | SUBPARTITION} partn_name WITH TABLE table_name [{INCLUDING | EXCLUDING} INDEXES] [{WITH | WITHOUT} VALIDATION] [ [NO]PARALLEL] [update_index_clause] [EXCEPTIONS INTO table_name]
```

Exchanges the data and index segments of a nonpartitioned table with a partitioned table, or of a partitioned table of one type to another partitioned type. The structure of the tables in the exchange must be identical, including the same primary key. All segment attributes (like tablespaces, logging, and statistics) of the current partitioned table, called *partn_name*, and the table it is being exchanged with, called *table_name*, are exchanged. Tables containing *LOB* columns will also exchange *LOB* data and index segments. Additional syntax details, which are not previously defined elsewhere in this list, follow:

WITH TABLE *table_name*

Defines the table that will exchange with the current partition or subpartition.

{*INCLUDING* | *EXCLUDING*} INDEXES

Exchanges local index partitions or subpartitions with the table index (on nonpartitioned tables) or the local index (on hash-partitioned tables), using the *INCLUDING INDEXES* clause. Alternately, marks all index partitions and subpartitions as well as regular indexes and partitioned indexes of the exchanged table to be marked with *UNUSABLE* status, using the *EXCLUDING INDEXES* clause.

{*WITH* | *WITHOUT*} VALIDATION

Returns errors when any rows in the current table fail to map into a partition or subpartition of the exchanged table, using the *WITH VALIDATION* clause. Otherwise, the *WITHOUT VALIDATION* clause may be included to skip checking of row mapping between the tables.

EXCEPTIONS INTO *table_name*

Places the *ROWIDs* of all rows violating a UNIQUE constraint (in DISABLE VALIDATE state) on the partitioned table. When omitted, Oracle assumes a table in the current schema called *EXCEPTIONS*. The *EXCEPTIONS* table is defined in the *utlexcpt.sql* and *utlexpt1.sql* scripts that ship with Oracle. Refer to the Oracle documentation if you need these scripts.

There are a couple of notes to remember about altering a partitioned table. First, altering a partition on a table that serves as the source for one or more materialized views requires that the materialized views be refreshed. Second, bitmap join indexes defined on the partitioned table being altered will be marked *UNUSABLE*. Third, several restrictions apply if the partitions (or subpartitions) are ever spread across tablespaces that use different block sizes. Refer to the Oracle documentation when attempting these sorts of alterations to a partitioned table.

In the next few examples, assume we are using the **orders** table partitioned as shown below:

CREATE TABLE orders

```
(order_number NUMBER,  
order_date DATE,  
cust_nbr NUMBER,  
price NUMBER,  
qty NUMBER,  
cust_shp_id NUMBER)
```

PARTITION BY RANGE(order_date)

```
(PARTITION pre_yr_2000 VALUES LESS THAN  
TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),  
PARTITION pre_yr_2004 VALUES LESS THAN  
TO_DATE('01-JAN-2004', 'DD-MON-YYYY'  
PARTITION post_yr_2004 VALUES LESS THAN  
MAXVALUE) );
```

The following statement will mark all of the local index partitions as *UNUSABLE* in the **orders** table for the **post_yr_2004** partition:

```
ALTER TABLE orders MODIFY PARTITION post_yr_2004  
UNUSABLE LOCAL INDEXES;
```

However, we've decided to now split the **orders** table partition **post_yr_2004** into two all new partitions, **pre_yr_2008** and **post_yr_2008**. Values that are now less than *MAXVALUE* will be stored in the **post_yr_2008** partition, while values less than '01-JAN-2008' will be stored in **pre_yr_2008**:

```
ALTER TABLE orders SPLIT PARTITION post_yr_2004
  AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
  INTO (PARTITION pre_yr_2008, PARTITION post_yr_2008);
```

Assuming that the **orders** table contained a **LOB** or a **VARRAY** column, we could further refine the alteration by including additional details for handling these columns, while also updating the global indexes as the operation completes:

```
ALTER TABLE orders SPLIT PARTITION post_yr_2004
  AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
  INTO
    (PARTITION pre_yr_2008
      LOB (order_desc) STORE AS (TABLESPACE order_tbspc_a1),
      PARTITION post_yr_2008)
      LOB (order_desc) STORE AS (TABLESPACE order_tbspc_a1) )
  UPDATE GLOBAL INDEXES;
```

Now, assuming the **orders** table has been grown at the upper end, lets merge together the partitions at the lower end:

```
ALTER TABLE orders
  MERGE PARTITIONS pre_yr_2000, pre_yr_2004
  INTO PARTITION yrs_2004_and_earlier;
```

After a few years more, we might want to even get rid of the oldest partition, or if not get rid of it, at least give it a better name:

```
ALTER TABLE orders DROP PARTITION yrs_2004_and_earlier;
ALTER TABLE orders RENAME PARTITION yrs_2004_and_earlier TO pre_yr_2004;
```

Finally, let's truncate a table partition, delete all of its data, and return the empty space for use by other objects in the tablespace:

```
ALTER TABLE orders
  TRUNCATE PARTITION pre_yr_2004
  DROP STORAGE;
```

As the examples illustrate, anything relating to partitioning and subpartitioning that can be created with the Oracle *CREATE TABLE* statement can later be changed, augmented, or cut down using the Oracle *ALTER TABLE* statement.

Organized tables: heaps, IOTs, and external tables

Oracle 10g allows powerful means of controlling the physical storage behavior of tables.

The most useful aspect of the *ORGANIZATION HEAP* clause is that you can now compress an entire table within Oracle. This is extremely useful for reducing disk storage costs in database environments with multi-terabyte tables. The following example creates the **orders** table in a compressed and logged heap along with a primary key constraint and storage details:

```
CREATE TABLE orders
(
  order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB,
  CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
ORGANIZATION HEAP
  COMPRESS LOGGING
PCTTHRESHOLD 2
STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
OVERFLOW STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
ENABLE ROW MOVEMENT;
```

To define the same table using an index-organized table based on the **order_date** column, we would use this syntax:

```
CREATE TABLE orders
(
  order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB,
  CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
ORGANIZATION HEAP
  INCLUDING order_date
PCTTHRESHOLD 2
STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
OVERFLOW STORAGE
  (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
ENABLE ROW MOVEMENT;
```

Finally, we will create an external table that stores our customer shipping information called **cust_shipping_external**. The code shown in bold is the *opaque_format_spec*:

```
CREATE TABLE cust_shipping_external
  (external_cust_nbr NUMBER(6),
  cust_shp_id      NUMBER,
  shipping_company VARCHAR2(25) )
ORGANIZATION EXTERNAL
  (TYPE oracle_loader
  DEFAULT DIRECTORY dataloader
  ACCESS PARAMETERS
    (RECORDS DELIMITED BY newline
    BADFILE 'upload_shipping.bad'
    DISCARDFILE 'upload_shipping.dis'
    LOGFILE 'upload_shipping.log'
    SKIP 20
    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY ""
    (client_id INTEGER EXTERNAL(6),
    shp_id CHAR(20),
    shipper CHAR(25) ) )
  LOCATION ('upload_shipping.ctl') )
REJECT LIMIT UNLIMITED;
```

Examining this example, we see that the external table type is *ORACLE_LOADER* and the default directory is *DATALOADER*. The example illustrates the fact that you define the metadata of the table within Oracle and then describe how that metadata references a data source outside of the Oracle database server itself.

Oracle XMLType and object-type tables

When an Oracle XMLType table is created, Oracle automatically stores the data in a *CLOB* column, unless you create an XML schema-based table. (For details on Oracle's XML support, see Oracle's XMLDB Developer's Guide.) The following code example first creates an XMLType table, **distributors**, with the implicit *CLOB* data storage, and followed by a second code example, **suppliers**, with a more sophisticated XML-schema definition:

```
CREATE TABLE distributors OF XMLTYPE;
CREATE TABLE suppliers OF XMLTYPE
XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
ELEMENT "vendors";
```

A key advantage of a table based on an XML schema is that you can create B-Tree indexes on them. In the following example, we create an index on **suppliercity**:

```
CREATE INDEX supplier-city-index
ON suppliers
(S:"XMLDATA"."ADDRESS"."CITY");
```

You may similarly create tables using a mix of standard and *XMLTYPE* columns. In this case, the *XMLTYPE* column may store its data as a *CLOB*, or it may store its data in an object-relational column of a structure determined by your specification. For example, we'll recreate the **distributors** table (this time with some added storage specifications) and the **suppliers** table with columns of both standard and *XMLTYPE*:

```
CREATE TABLE distributors
  (distributor_id NUMBER,
   distributor_spec XMLTYPE)
XMLTYPE distributor_spec
STORE AS CLOB
  (TABLESPACE tblspc_dist
   STORAGE (INITIAL 10M NEXT 5M)
   CHUNK 4000
   NOCACHE
   LOGGING);

CREATE TABLE suppliers
  (supplier_id NUMBER,
   supplier_spec XMLTYPE)
XMLTYPE supplier_spec STORE AS OBJECT RELATIONAL
  XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
  ELEMENT "vendors"
OBJECT IDENTIFIER IS SYSTEM GENERATED
OIDINDEX vendor_ndx TABLESPACE tblspc_xml_vendors;
```

When creating XML and object tables, you may refer to *inline_ref_constraint* and *table_ref_constraint* clauses. The syntax for an *inline_ref_constraint* clause is:

```
{SCOPE IS scope_table |
  WITH ROWID |
  [CONSTRAINT constraint_name] REFERENCES object [ (column_name) ]
  [ON DELETE {CASCADE | SET NULL} ]
  [constraint_state] }
```

The only difference between an inline reference constraint and a table reference constraint is that inline reference constraints operate at the column level and table reference constraints operate at the table level, respectively. (This is essentially the same behavior and coding rule as standard relational constraints like *PRIMARY KEY* or *FOREIGN KEY*.) The syntax for a *table_ref_constraint* follows:

```
{SCOPE FOR (ref_col | ref_attr) IS scope_table |
  REF (ref_col | ref_attr) WITH ROWID |
  [CONSTRAINT constraint_name] FOREIGN KEY (ref_col | ref_attr)
  REFERENCES object [ (column_name) ]
  [ON DELETE {CASCADE | SET NULL} ]
  [constraint_state] }
```

The *constraint_state* clause contains a number of options that have already been defined earlier in the Oracle *CREATE TABLE* statement. However, these options are only applied to the condition of the scope reference:

[NOT] DEFERRABLE

INITIALLY {IMMEDIATE | DEFERRED}

{ENABLE | DISABLE}

{VALIDATE | NOVALIDATE}

{RELY | NORELY}

EXCEPTIONS INTO *table_name*

USING INDEX {*index_name* | (*create_index_statement*) | *index_attributes*}

Object-type tables are useful for creating tables containing user-defined types. For example, the following code creates the **building_type** type:

```
CREATE TYPE OR REPLACE building_type AS OBJECT
  (building_name VARCHAR2(100),
  building_address VARCHAR2(200) );
```

We can then create a table called **offices_object_table** that contains the object and defines some of its characteristics, such as OID information. In addition, we'll create two more tables, based upon **building_type**, that reference the object type as an *inline_ref_constraint* and a *table_ref_constraint*, respectively:

```
CREATE TABLE offices_object_table
  OF building_type (building_name PRIMARY KEY)
OBJECT IDENTIFIER IS PRIMARY KEY;
CREATE TABLE leased_offices
  (office_nbr NUMBER,
  rent DEC(9,3),
  office_ref REF building_type
  SCOPE IS offices_object_table);
CREATE TABLE owned_offices
  (office_nbr NUMBER,
  payment DEC(9,3),
  office_ref REF building_type
  CONSTRAINT offc_in_bld REFERENCES offices_object_table);
```

In these examples, the *SCOPE IS* clause defines the *inline_ref_constraint*, while the *CONSTRAINT* clause defines the *table_ref_constraint*.

Oracle ALTER TABLE

When using the Oracle command *ALTER TABLE*, you are able to *ADD*, *DROP*, or *MODIFY* every aspect of each element of the table. For example, the syntax diagram shows that the method for adding or modifying an existing column includes its attributes, but we need to explicitly state that the *attributes* include any Oracle-specific extensions. So while the ANSI standard only lets you modify attributes such as *DEFAULT* or *NOT NULL*, as well as column-level constraints

assigned to the column, Oracle also allows you to alter any special characteristics like *LOB*, *VARRAY*, *NESTED TABLE*, index-organized table, *CLUSTER*, or *PARTITION* settings that might exist.

For example, the following code adds a new column to a table in Oracle, and also adds a new, unique constraint to that table:

```
ALTER TABLE titles
ADD subtitle VARCHAR2(32) NULL
CONSTRAINT unq_subtitle UNIQUE;
```

When a foreign key constraint is added to a table, the DBMS verifies that all existing data in the table meets that constraint. If they do not, the *ALTER TABLE* fails.



Any queries that use **SELECT *** return the new columns, even if this was not planned. Precompiled objects, such as stored procedures, can return any new columns if they use the **%ROWTYPE** attribute. Otherwise, a precompiled object may not return any new columns.

Oracle also allows multiple actions, such as *ADD* or *MODIFY*, to multiple columns to be performed by enclosing the actions within parentheses. For example, the following command adds several columns to a table with a single statement:

```
ALTER TABLE titles
ADD (subtitles VARCHAR2(32) NULL,
     year_of_copyright INT,
     date_of_origin DATE);
```

PostgreSQL

PostgreSQL supports the ANSI standards for *CREATE* and *ALTER TABLE* with a couple of extensions that enable you to quickly build a new table from existing table definitions. Following is the syntax for *CREATE TABLE*:

```
CREATE [LOCAL][[TEMP]ORARY TABLE table_name
(column_name datatype attributes [...])
| CONSTRAINT constraint_name [{NULL | NOT NULL}]
  {[UNIQUE] | [PRIMARY KEY (column_name [...])] | [CHECK (expression)]
  | REFERENCES reference_table (reference_column [...])
  [MATCH {FULL | PARTIAL | default}]
  [ON {UPDATE | DELETE}
  {CASCADE | NO ACTION | RESTRICT | SET NULL | SET DEFAULT value}]
  [(NOT) DEFERRABLE] [INITIALLY {DEFERRED | IMMEDIATE}] } [...]
| [table_constraint][,...]
[WITH[OUT] OIDE]
[INHERITS (inherited_table [...])]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[AS select_statement]
```


The PostgreSQL syntax for *ALTER TABLE* is:

```
ALTER TABLE [ONLY] table_name [*]  
  
[ADD [COLUMN] column_name datatype attributes [...] ]  
  
| [ALTER [COLUMN] column_name  
    {SET DEFAULT value | DROP DEFAULT | SET STATISTICS int}]  
  
| [RENAME [COLUMN] column_name TO new_column_name]  
  
| [RENAME TO new_table_name]  
  
| [ADD table_constraint]  
  
| [DROP CONSTRAINT constraint_name RESTRICT ]  
  
| [OWNER TO new_owner]
```

The parameters are as follows:

REFERENCES...MATCH...ON{UPDATE | DELETE} ...

Checks a value inserted into the column against the values of a column in another table. It also can be used as part of a *FOREIGN KEY* declaration. The *MATCH* options are *FULL*, *PARTIAL*, and the default where *MATCH* has no other keyword. Full match forces all columns of a multicolumn foreign key either to be NULL or to contain a valid value. The default allows mixed nulls and values. Partial matching is a valid syntax, but is not supported. The *REFERENCES* clause also allows several different behaviors to be declared for *ON DELETE* and/or *ON UPDATE* referential integrity as follows:

NO ACTION

Produces an error when the foreign key is violated (the default).

RESTRICT

A synonym for *NO ACTION*.

CASCADE

Sets the value of the referencing column to the value of the referenced column.

SET NULL

Sets the referencing column value to NULL.

SET DEFAULT *value*

Sets the referencing column to its declared default value or NULL, if no default value exists.

NOT DEFERRABLE [INITIALLY {DEFERRED | IMMEDIATE}]

The *DEFERRABLE* option of the *REFERENCES* clause tells PostgreSQL to defer evaluation of all constraints to the end of a transaction. *NOT DEFERRABLE* is the default behavior for the *REFERENCES* clause. Similar to the *DEFERRABLE* clause is the *INITIALLY* clause. Specifying *INITIALLY DEFERRED* checks constraints at the end of a transaction; *INITIALLY IMMEDIATE* checks constraints after each statement (the default).

FOREIGN KEY

Can be declared only as a table-level constraint and not as a column-level constraint. All options for the *REFERENCES* clause are supported as part of the *FOREIGN KEYS* clause. The syntax follows:

[*FOREIGN KEY* (*column_name* [...]) *REFERENCES*...]

WITH[OUT] OIDS

Specifies that rows of the new table should have automatically assigned object IDs (*WITH OIDS*) or not (*WITHOUT OIDS*). The default is *WITH OIDS*. *OIDS* are assigned based on a 32-bit counter. Once the counter is surpassed, the *OID* numbers cycle over and uniqueness can no longer be guaranteed. We recommend that you add a unique index to any *OID* columns you specify to avoid an error with duplicate *OIDS* on very large tables. A table with an inherited structure receives the *OID* setting of its parent table.

INHERITS inherited_table

Specifies a table or tables from which the table you are creating inherits all columns. The newly created table also inherits functions attached to tables higher in the hierarchy. If any inherited column appears more than once, the statement fails.

ON COMMIT {DELETE | PRESERVE} ROWS

Used only with temporary tables. This clause controls the behavior of the temporary table after records are committed to the table. *ON COMMIT DELETE ROWS* clears the temporary table of all rows after each commit. This is the default if the *ON COMMIT* clause is omitted. *ON COMMIT PRESERVE ROWS* saves the rows in the temporary table after the transaction has committed.

AS select_statement

Enables you to create and populate a table with data from a valid *SELECT* statement. The column names and datatypes do not need to be defined, since they are inherited from the query. The *CREATE TABLE...AS* statement has similar functionality to *SELECT...INTO*, but is more readable.

ONLY

Specifies that only the named table is affected by the *ALTER TABLE* statement and not any parent or subtables in a table hierarchy.

OWNER TO new_owner

Changes the owner of the table to the user identified by *new_owner*.

A PostgreSQL table cannot have more than 1,600 columns. However, you should limit the number of columns to well below 1,600 for performance reasons. For example:

CREATE TABLE distributors

```
(name VARCHAR(40) DEFAULT 'Thomas Nash Distributors',  
dist_id INTEGER DEFAULT NEXTVAL('dist_serial'),  
modtime TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```



Unique to PostgreSQL is the ability to create column-level constraints with multiple columns. Since PostgreSQL also supports standard table-level constraints, the ANSI-standard approach is still the recommended approach.

PostgreSQL's implementation of *ALTER TABLE* allows the addition of extra columns using the *ADD* keyword. Existing columns may have new default values assigned to them using *ALTER COLUMN...SET DEFAULT*, while *ALTER COLUMN...DROP DEFAULT* allows the complete erasure of a column-based default. In addition, new defaults may be

added to columns using the *ALTER* clause, but only newly inserted rows will be affected by such new default values. *RENAME* allows new names for existing columns and tables.

SQL Server

SQL Server offers a plethora of options when defining or altering a table, its columns, and its table-level constraints. Its *CREATE TABLE* syntax is:

```
CREATE TABLE table_name
({column_name datatype [ [DEFAULT default_value]
| {IDENTITY [(seed,increment) [NOT FOR REPLICATION]]]
| ROWGUIDCOL][NULL | NOT NULL]
| [{PRIMARY KEY | UNIQUE}
| [CLUSTERED | NONCLUSTERED]
| [WITH FILLFACTOR = inf] [ON {filegroup | DEFAULT}] ]
| [[FOREIGN KEY]
REFERENCES reference_table(reference_column[,...])
| ON {DELETE | UPDATE} {CASCADE | NO ACTION}]
| [NOT FOR REPLICATION]
| [CHECK [NOT FOR REPLICATION] (expression)
| [COLLATE collation_name]
| column_name AS computed_column_expression }
[,...]
| [table_constraint] [,...] )
[ON {filegroup | DEFAULT} ]
[TEXTIMAGE_ON {filegroup | DEFAULT} ]
```

The SQL Server version of *ALTER TABLE* is:

```
ALTER TABLE table_name
[ALTER COLUMN column_name new_datatype attributes {ADD | DROP} ROWGUIDCOL]
| [ADD [COLUMN] column_name datatype attributes [,...]
| [WITH CHECK | WITH NOCHECK] ADD table_constraint [,...]
| [DROP { [ CONSTRAINT ] constraint_name | COLUMN column_name }] [,...]
| [{ CHECK | NOCHECK } CONSTRAINT { ALL | constraint_name [,...] }]
| [{ ENABLE | DISABLE } TRIGGER { ALL | trigger_name [,...] }]
```

The parameters are as follows:

IDENTITY [(*seed, increment*)]

Creates and populates the column with a monotonically increasing number when applied to an integer column. The *IDENTITY* starts counting at the value of *seed* and increases by the value of *increment*. When either are omitted, the default is 1.

NOT FOR REPLICATION

Specifies that the values of an *IDENTITY* or *FOREIGN KEY* are not replicated to subscribing servers. This helps in situations in which different servers require the same table structures, but not the exact same data.

ROWGUIDCOL

Identifies a column as a global unique identifier, which ensures no two values are ever repeated across any number of servers. Only one such column may be identified per table. It does not, however, create the unique values itself. They must be inserted using the *NEWID* function.

DEFAULT

Applies to any column except those with a *TIMESTAMP* datatype or an *IDENTITY* property. The *DEFAULT* must be a constant value such as a character string or a number, and a system function such as *GETDATE()* or *NULL*.

PRIMARY KEY

Defines a primary key constraint for the table. The primary key declaration differs from the ANSI standard by allowing you to assign the clustered or nonclustered attributes on the primary key index, as well as a starting fillfactor. (Refer to the [CREATE/ALTER INDEX Statement](#) for more information.) The attributes of a primary key include:

CLUSTERED

Declares that the column or columns of the primary key set the physical sort order of the records in the table. *CLUSTERED* is the default when the clustered or nonclustered keywords are omitted.

NONCLUSTERED

Declares that the primary key index maintains pointers to all of the records of the table.

WITH FILLFACTOR = int

Declares that a percentage of space, *int*, should remain free on each data page when the table is created. SQL Server does not maintain *FILLFACTOR* over time. You should rebuild the index on a regular basis.

ON {filegroup | DEFAULT}

Specifies that the primary key is located either on the pre-existing, named filegroup or is assigned to the *DEFAULT* filegroup.

FOREIGN KEY

Checks values as they are inserted into the table against a column in another table in order to maintain referential integrity. Foreign keys are described in detail in [Chapter 2](#). A foreign key can only reference columns that are defined as a *PRIMARY KEY* or *UNIQUE* index on the referencing table. A referential action may be specified to take place on the *reference_table* when the record is deleted or updated, according to the following:

ON {DELETE | UPDATE}

Specifies that an action needs to happen in the local table when either (or both) an *UPDATE* or *DELETE* occur on the referenced table.

CASCADE

Specifies that any *DELETE* or *UPDATE* also takes place on the referring table to any records dependent on the value of the *FOREIGN KEY*.

NO ACTION

Specifies that no action occur on the referring table when a record on the current table is deleted or updated.

NOT FOR REPLICATION

Specifies that an *IDENTITY* property should not be enforced when the data is replicated from another database. This ensures that data from the published server are not assigned new identity values.

CHECK

Ensures that a value inserted into the specified column of the table is a valid value based on the *CHECK* expression. For example, the following shows a table with two column-level *CHECK* constraints.

CREATE TABLE people

```
(people_id CHAR(4)
    CONSTRAINT pk_dist_id PRIMARY KEY CLUSTERED
    CONSTRAINT ck_dist_id CHECK (dist_id LIKE '
[A-Z][A-Z][A-Z][A-Z]),
people_name VARCHAR(40) NULL,
people_addr1 VARCHAR(40) NULL,
people_addr2 VARCHAR(40) NULL,
city VARCHAR(20) NULL,
state CHAR(2) NULL
    CONSTRAINT def_st DEFAULT ("CA")
    CONSTRAINT chk_st REFERENCES states(state_ID),
zip CHAR(5) NULL
    CONSTRAINT ck_dist_zip
CHECK(zip LIKE '[0-9][0-9][0-9][0-9]'),
phone CHAR(12) NULL,
sales_rep empid NOT NULL DEFAULT USER)
```

GO

The *CHECK* constraint on the **people_id** ensures an all-alphabetic ID, while the one on *zip* ensures an all-numeric value. The *REFERENCES* constraint on *state* performs a look-up on the **states** table. The *REFERENCES* constraint is essentially the same as a *CHECK* constraint, except that it derives its list of acceptable values from the values stored in another column. This example illustrates how column-level constraints are named using the *CONSTRAINT constraint_name* syntax.

COLLATE

Allows programmers to change, on a column-by-column basis, the sort order and character set that is used by the column.

TEXTIMAGE_ON {filegroup | DEFAULT}

Controls the placement of **text**, **ntext**, and **image** columns allowing you to place *LOB* data on the pre-existing *filegroup* of your choice. When omitted, these columns are stored in the default filegroup with all other tables and database objects.

WITH [NO]CHECK

Tells SQL Server whether the data in the table should be validated against any newly added constraints or keys. When constraints are added using *WITH NOCHECK*, the query optimizer ignores them until they are enabled via *ALTER TABLE table_name CHECK CONSTRAINT ALL*. When constraints are added using *WITH CHECK*, the constraints are checked immediately against all data already in the table.

[NO]CHECK CONSTRAINT

Enables an existing constraint with *CHECK CONSTRAINT* or disables one with *NOCHECK CONSTRAINT*.

```
{ENABLE | DISABLE} TRIGGER { ALL | trigger_name [,...] }
```

Enables or disables the specified trigger or triggers, respectively. All triggers on the table may be enabled or disabled by substituting the keyword *ALL* for the table name, as in *ALTER TABLE employee DISABLE TRIGGER ALL*. You may, alternately, disable or enable a single *trigger_name* or more than one trigger by placing each *trigger_name* in a comma-delimited list.

SQL Server allows any column-level constraint to be named by specifying *CONSTRAINT constraint_name...*, and then the text of the constraint. Several constraints may be applied to a single column, as long as they are not mutually exclusive (for example, *PRIMARY KEY* and *NULL*).

SQL Server also allows a local temporary table to be created, but not using ANSI syntax. A local temporary table, which is stored in the **tempdb** database, requires a prefix of a single pound sign (#) to the name of the table. The local temporary table is usable by the person or process that created it, and is deleted when the person logs out or the process terminates. A global temporary table, which is usable to all people and processes that are currently logged in, can be established by prefixing two pound signs (##) to the name of the table. The global temporary table is deleted when its process terminates or its creator logs out.

SQL Server also allows the creation of tables with columns that contain a computed value. Such a column does not actually contain data. Instead, it is a virtual column containing an expression using other columns already in the table. For example, a computed column could have an expression, such as *order_cost AS (price * qty)*. Computed columns also can be constants, functions, variables, noncomputed columns, or any of these combined with each other using operators.

Any of the column-level constraints shown earlier also may be declared at the table level. That is, *PRIMARY KEY* constraints, *FOREIGN KEY* constraints, *CHECK* constraints, and others may be declared after all the columns have been defined in the *CREATE TABLE* statement. This is very useful for constraints that cover more than one column. For example, a column-level *UNIQUE* constraint can be applied only to that column. However, declaring the constraint at the table level allows it to span several columns. Here is an example of both column-level and table-level constraints:

-- Creating a column-level constraint

```
CREATE TABLE favorite_books
```

```
(isbn      CHAR(100)  PRIMARY KEY NONCLUSTERED,  
 book_name VARCHAR(40) UNIQUE,  
 category  VARCHAR(40) NULL,  
 subcategory VARCHAR(40) NULL,  
 pub_date  DATETIME  NOT NULL,  
 purchase_date DATETIME NOT NULL)
```

```
GO
```

-- Creating a table-level constraint

```
CREATE TABLE favorite_books
```

```
(isbn      CHAR(100)  NOT NULL,  
 book_name VARCHAR(40) NOT NULL,  
 category  VARCHAR(40) NULL,  
 subcategory VARCHAR(40) NULL,  
 pub_date  DATETIME  NOT NULL,  
 purchase_date DATETIME NOT NULL,  
 CONSTRAINT pk_book_id PRIMARY KEY NONCLUSTERED (isbn)  
 WITH FILLFACTOR=70,
```

```
CONSTRAINT unq_book UNIQUE CLUSTERED (book_name, pub_date)
```

```
GO
```

These two commands provide nearly the same results, except that the table-level *UNIQUE* constraint has two columns, whereas only one column is included in the column-level *UNIQUE* constraint.

The following example adds a new *CHECK* constraint to a table, but does not check to ensure that the existing values in the table pass the constraint:

```
ALTER TABLE favorite_book WITH NOCHECK
```

```
ADD CONSTRAINT extra_check CHECK (ISBN > 1)
```

```
GO
```

In this example, we further add a new column with an assigned *DEFAULT* value that is placed in each existing record in the table:

```
ALTER TABLE favorite_book ADD reprint_nbr INT NULL
```

```
CONSTRAINT add_reprint_nbr DEFAULT 1 WITH VALUES
```

```
GO
```

```
-- Now, disable the constraint
```

```
ALTER TABLE favorite_book NOCHECK CONSTRAINT add_reprint_nbr
```

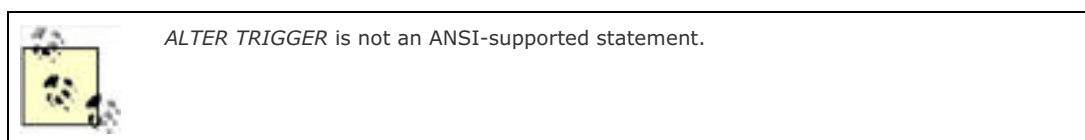
```
GO
```

See Also

```
CREATE SCHEMA  
DROP
```

CREATE/ALTER TRIGGER Statement

A *trigger* is a special kind of stored procedure that fires automatically (hence the term "trigger") when a specific data-modification statement is executed against a table. The trigger is directly associated with the table and is considered a dependent object. For example, you might want all of the **part_numbers** in the **sales** table to be updated when a **part_number** is changed in the **products** table, thus ensuring that part numbers are always in sync.



Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {DELETE | INSERT | UPDATE [OF column [,...]]}
ON table_name
[REFERENCING {OLD {[ROW] | TABLE} [AS] old_name | NEW
{ROW | TABLE} [AS] new_name}] [FOR EACH { ROW | STATEMENT }]
[WHEN (conditions)]
[BEGIN ATOMIC]
code_block
[END]
```

Keywords

CREATE TRIGGER *trigger_name*

Creates a trigger of *trigger_name* and associates it with a specific table.

BEFORE | *AFTER*

Declares that the trigger logic is fired either *BEFORE* or *AFTER* the data manipulation operation that invoked the trigger. *BEFORE* triggers perform their operations before the *INSERT*, *UPDATE*, or *DELETE* operation occurs, allowing you to do dramatic things like circumvent the data manipulation operation altogether. *AFTER* triggers fire after the data manipulation operation has completed, but are useful for after-the-fact operations like recalculating running totals.

DELETE | INSERT | UPDATE [OF *column* [,...]]

Defines the data manipulation operation that causes the trigger to fire: *DELETE* statements, *INSERT* statements, or *UPDATE* statements. You may optionally chose which columns will trigger an update trigger using *UPDATE OF column* [,...]. If an update occurs on any columns not in the column list, the trigger will not fire.

ON *table_name*

Declares the pre-existing table that the trigger is dependent on.

REFERENCING {OLD {[ROW] | TABLE} [AS] *old_name* | NEW {ROW | TABLE} [AS] *new_name*}

Enables aliasing for the old or new *ROW* or *TABLE* acted upon by the trigger. Although the syntax shows the options as exclusive, you may have up to four aliasing references: one for old row, one for old table, one for new row, and one for new table. The alias *OLD* refers to the data contained in the table or row before the data manipulation operation that fired the trigger, while the alias *NEW* refers to the data that would be contained in the table or row after the data manipulation operation that fired the trigger. Note that the syntax indicates that *ROW* is optional, but *TABLE* is not. (That is, *OLD ROW AS* is the same as *OLD AS*, but for *TABLE*, the only valid option is *OLD TABLE AS*.) *INSERT* triggers do not have an *OLD* context, while *DELETE* triggers do not have a *NEW* context. The keyword *AS* is noise and may be omitted. If the *REFERENCING* clause specifies either *OLD ROW* or *NEW ROW*, then the *FOR EACH ROW* clause is required.

FOR EACH { ROW | STATEMENT }

Tells the database to apply the trigger for each row in the table that has changed (*ROW*) or for each SQL

statement issued against the table (*STATEMENT*). For example, a single *UPDATE* statement will update the salaries of 100 employees. If you specify *FOR EACH ROW*, the trigger will execute 100 times. If you specify *FOR EACH STATEMENT*, the trigger will execute only once.

WHEN (*conditions*)

Allows you to define additional criteria for a trigger. For example, you might have a trigger called *DELETE employee*, which will fire whenever an employee is deleted. When a trigger fires, if the search conditions contained in the *WHEN* clause evaluate to TRUE, the trigger action will fire. Otherwise, the trigger will not fire.

BEGIN ATOMIC | *code_block* | END

The ANSI standard requires that the *code_block* should contain only one SQL statement or, for multiple SQL statements, put them in a *BEGIN* and *END* block.

Rules at a Glance

Triggers, by default, fire once at the *statement level*. That is, a single *INSERT* statement might insert 500 rows into a table, but an insert trigger on that table fires only one time. Some vendors allow a trigger to fire for each row of the data-modification operation. So, a statement that inserts 500 rows into a table that has a row-level insert trigger fires 500 times, once for each inserted row.

In addition to being associated with a specific data-modification statement (*INSERT*, *UPDATE*, or *DELETE*) on a given table, triggers are associated with a specific *time* of firing. In general, triggers can fire *BEFORE* the data-modification statement is processed, *AFTER* it is processed, or (when supported by the vendor) *INSTEAD OF* processing the statement. Triggers that fire before or instead of the data-modification statement do not see the changes that the statement renders, while those that fire afterwards can see and act upon the changes that the data-modification statement renders.

Triggers make use of two *pseudotables*. They are pseudotables in the sense that they are not declared with a *CREATE TABLE* statement, but they exist logically on the database. The pseudotables have different names on the different platforms, but we'll call them **Before** and **After** here. They are structured exactly the same as the table a trigger is associated with. However, they contain snapshots of the table's data. The **Before** table contains a snapshot of all the records in the table before the trigger fired, while the **After** table contains a snapshot of all the records in the table after the event occurred that fired the trigger. You can then use comparison operations to compare the data in the table before and after the event fired the trigger to determine exactly what you want to happen.

Here is an Oracle *BEFORE* trigger that uses the *OLD* and *NEW* pseudotables to compare values. (By way of comparison, SQL Server uses the *DELETED* and *INSERTED* pseudotables in the same way, while DB2 requires that you provide a *REFERENCING* clause to give names to the pseudotables.) This trigger creates an audit record before changing an employee's pay record:

```
CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
WHEN (new.emp_salary <> old.emp_salary)
BEGIN
    INSERT INTO employee_audit
    VALUES ('old', :old.emp_id, :old.emp_salary, :old.emp_ssn);
END;
```

You can also take advantage of capabilities within each database platform to make your triggers more powerful and easier to program. For example, Oracle has a special *IF...THEN* clause for use just in triggers. This *IF...THEN* clause takes the form: *IF DELETING THEN*, *IF INSERTING THEN*, or *IF UPDATING THEN*. The following example builds an Oracle *DELETE* and *UPDATE* trigger that uses the *IF DELETING THEN* clauses:

```
CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
BEGIN
    IF DELETING THEN
```

```
INSERT INTO employee_audit
VALUES ('DELETED', :old.emp_id, :old.emp_salary, :old.emp_ssn);

ELSE

INSERT INTO employee_audit
VALUES ('UPDATED', :old.emp_id, :new.emp_salary, :old.emp_ssn);

END IF;

END;
```

This SQL Server example adds a new table called **contractor** to the database. All records in the **employee** table that indicate that the employee is a contractor were moved into the **contractor** table. Now all new employees inserted into the **employee** table will go into the **contractor** table instead through an *INSTEAD OF INSERT* trigger:

```
CREATE TRIGGER if_emp_is_contractor
INSTEAD OF INSERT ON employee
BEGIN
INSERT INTO contractor
SELECT * FROM inserted WHERE status = 'CON'
INSERT INTO employee
SELECT * FROM inserted WHERE status = 'FTE'
END
GO
```

Adding triggers to a table that already has data in it does not cause the triggers to fire. A trigger will fire only for data-modification statements declared in the trigger definition that occur after the trigger is created.

A trigger generally ignores structural changes to tables, such as an added column or an altered datatype on an existing column, after the trigger has been created unless a table modification directly interferes with the operation of the trigger. For example, adding a new column to a table will result in any existing trigger ignoring the new column, except for *UPDATE* triggers. On the other hand, removing a column from a table that is used by a trigger will cause the trigger to fail every time it executes.

Programming Tips and Gotchas

One of the key programming issues associated with triggers is the inappropriate and uncontrolled use of *nested triggers* and *recursive triggers*. A nested trigger is a trigger that invokes a data-manipulation operation that causes other triggers to fire. For example, assume we have three tables, **T1**, **T2**, and **T3**. Table **T1** has a *BEFORE INSERT* trigger that inserts a record into table **T2**. Table **T2** also has a *BEFORE INSERT* trigger that inserts a record into table **T3**. Although this is not necessarily a bad thing if your logic is well considered and fully thought out, it does introduce two problems. First, an insert into table **T1** requires many more I/O operations and transactions than a simple *INSERT* statement against table **T1**. Second, you can get yourself into hot water if, in fact, table **T3** performs an *INSERT* operation against table **T1**. In a case like that, you may have a looping trigger process that can consume all available disk space and even shut down the server.

Recursive triggers are triggers that can fire themselves; for example an *INSERT* trigger that performs an *INSERT* against its own base table. If the procedural logic within the *code_body* is not properly constructed, you can cause a looping trigger error. However, recursive triggers often require setting a special configuration flag before they can be used on the various database platforms.

DB2

DB2 supports a superset of the ANSI SQL standard for the *CREATE TRIGGER* statement. It *does not* support an *ALTER TRIGGER* statement. The *CREATE TRIGGER* syntax is:

```
CREATE TRIGGER trigger_name
```

```
{NO CASCADE BEFORE | AFTER | INSTEAD OF}  
{[DELETE] | [INSERT] | [UPDATE [OF (column [,...]) ]]}  
ON table_name  
[REFERENCING  
  {OLD[_TABLE] [AS] old_name | NEW[_TABLE]  
  [AS] new_name} DEFAULTS NULL]  
[FOR EACH { ROW | STATEMENT }]  
MODE DB2SQL  
[WHEN (conditions)  
code_block
```

where:

NO CASCADE BEFORE

Declares that the trigger action is applied *before* the data-modification operation is applied to the table. This clause also declares that the trigger action will not cause any other triggers to fire.

ON *table_name*

Specifies the pre-existing table or view that the trigger is associated with. Triggers can be declared on regular tables, but not on temporary tables, nicknames, catalog tables, or materialized query tables. When creating triggers on a view, only *INSTEAD OF* triggers are allowed. The view must be a non-typed view that does not use the *WITH CHECK OPTION* clause. The data-modification operation applied to the view should be translated against the base table to determine whether it will fire the trigger or not.

OLD[_TABLE] [AS] *old_name* | NEW[_TABLE] [AS] *new_name*

Defines the alias for the ANSI *before* and *after* pseudotables using *OLD_TABLE* and *NEW_TABLE*, respectively. If you leave off the *_TABLE* keyword, then DB2 assigns the trigger operation as row-level, and allows you to access the row (singular) data prior to triggering the SQL operation. Adding in the *_TABLE* keyword declares that the trigger operation allows you to reference the set (plural) of row data prior to triggering the SQL operation. *OLD* and *OLD_TABLE* may only be used on *DELETE* or *UPDATE* triggers. Conversely, *NEW* and *NEW_TABLE* can only be used on *INSERT* and *UPDATE* triggers. *OLD* and *NEW* may only be specified once each. *OLD_TABLE* and *NEW_TABLE* may only be specified once each and not on *NO CASCADE BEFORE* triggers.

DEFAULTS

Inserts default values where they have been created.

FOR EACH { ROW | STATEMENT }

Specifies that the trigger action applies either at the row level or the statement level. The *FOR EACH STATEMENT* clause is only allowed on *AFTER* triggers.

MODE DB2SQL

Declares that the mode of the trigger is DB2 SQL, the only mode currently supported.

WHEN (*conditions*)

Meets the ANSI SQL standard with the exception that this clause is not allowed on DB2 *INSTEAD OF* triggers.

DB2 does not prevent a new trigger from being created that has the same definition as an older trigger. Thus, it's possible to have two (or more) *AFTER DELETE* triggers on a single table. In a situation like this, DB2 will always fire the

triggers in the order they were created.

Be careful declaring a *BEFORE DELETE* trigger on a table with a self-referencing constraint and a delete rule of *CASCADE*. Such a trigger may produce results that you did not expect!



DB2 allows the *COMMENT ON object_type* statement, so that you can provide extra commentary on any supported DB2 object. For example, you can issue the *COMMENT ON TRIGGER trigger_name* statement to add comments on any trigger you add to the database. You can use this on tables, views, etc.

DB2 does not specifically drop the triggers associated with a table or view when the table or view itself is dropped. Consequently, you should first issue a *DROP TRIGGER* statement followed by a *DROP TABLE* statement to avoid leaving orphaned triggers behind. Conversely, triggers are not reinstated if a table is dropped and then recreated. The triggers should be recreated as well.

BEFORE and *AFTER* triggers are allowed on typed tables at any level of a typed-table hierarchy. Triggers may activate on rows of subtables. If a SQL statement activates multiple triggers, the triggers will be executed in their creation order, even if they are attached to different tables in the typed-table hierarchy.

MySQL

Not supported.

Oracle

Oracle supports the ANSI standard for *CREATE TRIGGER*, with several additions and variations:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{ [{object_event] [database_event] [...] ON {DATABASE | schema.SCHEMA} } |
  { [DELETE] [OR] [INSERT] [OR] [UPDATE [OF column [,...]]] [...] }
  ON {table_name | [NESTED TABLE column_name OF] view_name}
  [REFERENCING {[OLD [AS] old_name] [NEW [AS] new_name]
  [PARENT [AS] parent_name]}
  [FOR EACH ROW] }
[WHEN (conditions)]
code_block
```

The syntax for *ALTER TRIGGER*, which allows you to rename, enable, or disable a trigger without dropping and recreating it, is shown below:

```
ALTER TRIGGER trigger_name
{ {ENABLE | DISABLE} | RENAME TO new_name |
  COMPILE [compiler_directives] [DEBUG] [REUSE SETTINGS]}
```

where:

OR REPLACE

Recreates an existing trigger by assigning a new definition to an existing trigger of *trigger_name*.

object_event

In addition to the standard data-modification events, Oracle allows triggers to fire based on object events. *object_event* operations may be paired with *BEFORE* and *AFTER* keywords. An *object_event* fires the trigger whenever such an event occurs, according to the following keywords:

ALTER

Fires whenever an *ALTER* statement (except *ALTER DATABASE*) is issued.

ANALYZE

Fires whenever Oracle validates the structure of a database object, or collects or deletes statistics on an index.

ASSOCIATE STATISTICS

Fires whenever Oracle associates a statistics type with a database object.

AUDIT

Fires whenever Oracle tracks a SQL statement or operation against a schema object.

COMMENT

Fires whenever an Oracle comment is added in the data dictionary to a database object.

DDL

Fires whenever Oracle encounters any *object_event* in this list.

DISASSOCIATE STATISTICS

Fires whenever Oracle disassociates a statistics type from a database object.

DROP

Fires whenever a *DROP* statement erases a database object from the data dictionary.

GRANT

Fires whenever a user grants privileges or roles to another user or role.

NOAUDIT

Fires whenever the *NOAUDIT* statement causes Oracle to stop tracking SQL statements or operations against schema objects.

RENAME

Fires whenever the *RENAME* statement changes the name of a database object.

REVOKE

Fires whenever a user revokes privileges or roles from another user or role.

TRUNCATE

Fires whenever a *TRUNCATE* statement is issued against a table or cluster.

database_event

In addition to the standard data-modification events, Oracle allows triggers to fire based on database events. *Database_event* operations may be paired with *BEFORE* and *AFTER* keywords. The list of allowable *database_event* keywords is:

LOGON

Fires whenever a client application logs on to the database. Valid for *AFTER* triggers only.

LOGOFF

Fires whenever a client application logs off of the database. Valid for *BEFORE* triggers only.

SERVERERROR

Fires whenever a server error message is logged. Valid for *AFTER* triggers only.

SHUTDOWN

Fires whenever an instance of the database is shut down. Valid for *BEFORE* triggers only with the *ON DATABASE* clause only.

STARTUP

Fires whenever an instance of the database is opened. Valid for *AFTER* triggers with the *ON DATABASE* clause.

SUSPEND

Fires whenever a server error causes a transaction to suspend. Valid for *AFTER* triggers only.

ON {DATABASE | schema.SCHEMA}

Declares that the trigger fires whenever any database user invokes a triggering event with *ON DATABASE*. The trigger then fires for events occurring anywhere in the entire database. Otherwise, *ON schema.SCHEMA* declares that the trigger fires whenever a user connected as *schema* invokes a triggering event. The trigger then fires for events occurring anywhere in the current schema.

ON [NESTED TABLE column_name OF] view_name

Declares that the trigger fires only if the data-manipulation operation applies to the column(s) of the view called *view_name*. The *ON NESTED TABLE* clause is compatible only with *INSTEAD OF* triggers.

REFERENCING PARENT [AS] parent_name

Defines the alias for the current row of the parent table (i.e., supertable). Otherwise, identical to the ANSI standard.

ENABLE

Enables a deactivated trigger when used with *ALTER TRIGGER*. You may alternately use the statement *ALTER TABLE table_name ENABLE ALL TRIGGERS*.

DISABLE

Disables an activated trigger when used with *ALTER TRIGGER*. You may alternately use the statement *ALTER TABLE table_name DISABLE ALL TRIGGERS*.

RENAME TO *new_name*

Renames the trigger to *new_name*, though the state of the trigger remains unchanged when used with *ALTER TRIGGER*.

COMPILE [DEBUG] [REUSE SETTINGS]

Compiles a trigger, whether valid or invalid, and all the objects that the trigger depends on. If any the objects are invalid, then the trigger is invalid. If all of the objects are valid, including the *code_body* of the trigger, then the trigger becomes valid. The subclauses of the *COMPILE* clause are:

DEBUG

Tells the PL/SQL compiler to generate and store the extra information for use by the PL/SQL debugger.

REUSE SETTINGS

Tells Oracle to retain all compiler switch settings, which can save significant time during the compile process.

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive* = '*value*'. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

When referencing values in the *OLD* and *NEW* pseudotables, the values must be prefaced with a colon (:), except in the trigger's *WHEN* clause, where no colons are used. In this example, we'll call a procedure in the *code_body* and use both *:OLD* and *:NEW* values as arguments:

```
CREATE TRIGGER scott.sales_check
BEFORE INSERT OR UPDATE OF ord_id, qty ON scott.sales
FOR EACH ROW
WHEN (new.qty > 10)
CALL check_inventory(:new.ord_id, :new.qty, :old.qty);
```

Multiple trigger types may be combined into a single trigger command *if* they are of the same level (row or statement) and they are on the same table. When triggers are combined in a single statement, the clauses *IF INSERTING THEN*, *IF UPDATING THEN*, and *IF DELETING THEN* may be used in the PL/SQL block to break the code logic into distinct segments. An *ELSE* clause also can be used in this structure.

Following is an example of a *database_event*-style trigger:

```
CREATE TRIGGER track_errors
AFTER SERVERERROR ON DATABASE
BEGIN
IF (IS_SERVERERROR (04030))
THEN INSERT INTO errors ('Memory error');
ELSE (IS_SERVERERROR (01403))
THEN INSERT INTO errors ('Data not found');
END IF;
END;
```

In another example, we can create a trigger that is *SCHEMA*-wide in scope:

```
CREATE OR REPLACE TRIGGER create_trigger
AFTER CREATE ON scott.SCHEMA
BEGIN
    RAISE_APPLICATION_ERROR (num => -20000, msg =>
        'Scott created an object');
END;
```

PostgreSQL

PostgreSQL's implementation of *CREATE TRIGGER* offers a subset of features found in the ANSI standard. On PostgreSQL, a trigger may fire *BEFORE* a data-modification operation is attempted on a record and before any constraints are fired or it may fire *AFTER* a data-manipulation operation fires (and after constraints have been checked), making all operations involved in the transaction visible to the trigger. Finally, the trigger may fire *INSTEAD OF* the data-modification operation and completely replace the *INSERT*, *UPDATE*, or *DELETE* statement with some other behavior. The *CREATE TRIGGER* syntax is:

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER }
{ {[DELETE] [OR | ,] [INSERT] [OR | ,] [UPDATE]} [OR ...] }
ON table_name
FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE function_name (parameters)
```

The parameters are:

OR

Declares an additional trigger action. The *OR* keyword is a synonym for the comma delimiter.

FOR EACH ROW

Explicitly declares that the trigger acts on the row level. This is also the default. Although PostgreSQL does not reject the *STATEMENT* keyword, it does not enforce statement-level triggers.

EXECUTE PROCEDURE *function_name* (*parameters*)

Executes a previously defined function (created using *CREATE FUNCTION*) rather than a block of procedural code. (PostgreSQL does not have its own procedural language.)

The following is an example of a PostgreSQL *BEFORE* trigger that checks at a row level to ensure that the specified distributor code exists in the **distributors** table before inserting or updating a row in the **sales** table:

```
CREATE TRIGGER if_dist_exists
BEFORE INSERT OR UPDATE ON sales
FOR EACH ROW
EXECUTE PROCEDURE check_primary_key
    ('dist_id', 'distributors', 'dist_id');
```


BEFORE and *AFTER* triggers are supported by the ANSI standard. *INSTEAD OF* triggers on PostgreSQL completely skip the data-modification operation that triggered them in favor of code that you substitute for the data-modification operation.

SQL Server

SQL Server supports the basics of the ANSI standard with the addition of the *INSTEAD OF* trigger type and a column change check. It does not support the *REFERENCING* or *WHEN* clauses. Its syntax is:

```
{CREATE | ALTER} TRIGGER trigger_name ON table_name
[WITH ENCRYPTION]
{FOR | AFTER | INSTEAD OF} {[DELETE] [,] [INSERT] [,] [UPDATE]}
[WITH APPEND]
[NOT FOR REPLICATION]
AS
[IF UPDATE(column) [{AND | OR} UPDATE(column)][... ]
code_block
```

where:

```
{CREATE | ALTER} TRIGGER trigger_name
```

Creates a new trigger of *trigger_name* or alters an existing trigger of *trigger_name* by adding or changing trigger properties or the trigger *code_block*. When altering an existing trigger, the permissions and dependencies of the existing trigger are maintained.

```
ON table_name
```

Declares the table or view that the trigger is dependent on. Views may have *INSTEAD OF* triggers defined on them, as long as they are updateable and do not have the *WITH CHECK* clause on them.

```
WITH ENCRYPTION
```

Encrypts the text of the *CREATE TRIGGER* statement as shown in the **syscomments** table. This option is useful to protect important intellectual property. *WITH ENCRYPTION* prevents the trigger from use in a SQL Server replication scheme.

```
FOR | AFTER | INSTEAD OF
```

Tells SQL Server when the trigger should fire. (*FOR* and *AFTER* are synonyms.) *AFTER* specifies that the trigger fire only after the triggering data-modification statement (and any cascading actions and constraint checks) have completed successfully. The *INSTEAD OF* trigger is similar to the ANSI *BEFORE* trigger in that the code of the trigger may completely replace the data-modification operation. It specifies that the trigger is executed instead of the data-modification statement that fired the trigger. *INSTEAD OF DELETE* triggers cannot be used when there is a cascading action on the delete. Only *INSTEAD OF* triggers can access *TEXT*, *NTEXT*, or *IMAGE* columns.

```
WITH APPEND
```

Adds an additional trigger of an existing type to a table or view. This clause is supported for backward compatibility with earlier versions of the product and can be used only with *FOR* triggers.

```
NOT FOR REPLICATION
```

Prevents data-manipulation operations invoked through SQL Server's built-in replication engine from firing the trigger.

IF UPDATE(*column*) [{AND | OR} UPDATE(*column*)]...

Allows you to choose the specific columns that fire the trigger. A column-specific trigger fires only on *UPDATE* and *INSERT* operations, but not on *DELETE* operations. If an *UPDATE* or *INSERT* occurs on any columns not in the column list, the trigger will not fire.

SQL Server allows multiple triggers for a given data-manipulation operation on a table or view. Thus, three *UPDATE* triggers are possible on a single table. Multiple *AFTER* triggers are possible on a given table. Their specific order is undefined, though the first and last triggers can be explicitly declared using the *sp_settriggerorder* system stored procedure. Only one *INSTEAD OF* trigger is possible per *INSERT*, *UPDATE*, or *DELETE* statement on a given table.

In SQL Server, any combination of triggers is made possible in a single trigger definition statement by separating each option with a comma. (When doing so, the same code fires for each statement in the combination definition.)

SQL Server implicitly fires in the *FOR EACH STATEMENT* style of the ANSI standard.

SQL Server instantiates two important pseudotables when a trigger is fired: **deleted** and **inserted**. They are equivalent to the **before** and **after** pseudotables, respectively, described earlier in the ANSI [Rules at a Glance](#) section. These tables are identical in structure to the table on which the triggers are defined, except that they contain the old data before the data-modification statement fired (*deleted*) and the new values of the table after the data-modification statement fired (*inserted*).

The *AS IF UPDATE(column)* clause tests specifically for *INSERT* or *UPDATE* actions on a given column or columns, in the way the ANSI statement uses the *UPDATE(column)* syntax. Specify multiple columns by adding separate *UPDATE(column)* clauses after the first. Follow the *AS IF UPDATE(column)* clause with a Transact-SQL *BEGIN...END* block to allow the trigger to fire multiple Transact-SQL operations. This clause is functionally equivalent to the *IF...THEN...ELSE* operation.

In addition to intercepting data modification statements as shown in the ANSI SQL example, SQL Server allows you to perform other sorts of actions when a data modification operation occurs. In the following example, we've decided that the table **sales_archive_2002** is off limits and that anyone who attempts to alter data in these tables is to be notified of that restriction:

```
CREATE TRIGGER archive_trigger
ON sales_archive_2002
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10, 'No changes allowed to this table')
GO
```

SQL Server *does not* allow the following statements within the Transact-SQL *code_block* of a trigger: *ALTER*, *CREATE*, *DROP*, *DENY*, *GRANT*, *REVOKE*, *LOAD*, *RESTORE*, *RECONFIGURE*, or *TRUNCATE*. In addition, it does not allow any *DISK* statements or the *UPDATE STATISTICS* command.

SQL Server also allows triggers to fire recursively using the *recursive triggers* setting of the **sp_dboption** system stored procedure. Recursive triggers, by their own action, cause themselves to fire again. For example, if an *INSERT* trigger on table **T1** performs an *INSERT* operation on table **T1**, it might perform a recursive operation. Since recursive triggers can be dangerous, this functionality is disabled by default.

Similarly, SQL Server allows *nested triggers* up to 32 levels deep. If any one of the nested triggers performs a *ROLLBACK* operation, no further triggers execute. An example of nested triggers is a trigger on table **T1** firing an operation against table **T2**, which also has a trigger that fires an operation against table **T3**. The triggers cancel if an infinite loop is encountered. Nested triggers are enabled with the nested triggers setting of the system stored procedure **sp_configure**. If nested triggers are disabled, recursive triggers are disabled as well, despite the recursive triggers setting of **sp_dboption**.

In the following example, we want to reroute the user activity that occurs on the **people** table, especially update transactions, so that changes to records in the **people** table are instead written to the **people_reroute** table. (A more elaborate version of the **people** table is shown in [SQL Server](#) in the [CREATE/ALTER TABLE Statement](#) section.) Our update trigger will record any changes to columns 2, 3, or 4 of the **people** table and write it to the **people_reroute** table. The trigger will also record which user issued the update transaction and the time of the transaction:

```
CREATE TABLE people
(
    people_id CHAR(4),
    people_name VARCHAR(40),
    people_addr VARCHAR(40),
    city VARCHAR(20),
    state CHAR(2),
    zip CHAR(5),

```

```
phone CHAR(12),
sales_rep empid NOT NULL)
GO
CREATE TABLE people_reroute
(reroute_log_id UNIQUEIDENTIFIER DEFAULT NEWID( ),
reroute_log_type CHAR (3) NOT NULL,
reroute_people_id CHAR(4),
reroute_people_name VARCHAR(40),
reroute_people_addr VARCHAR(40),
reroute_city VARCHAR(20),
reroute_state CHAR(2),
reroute_zip CHAR(5),
reroute_phone CHAR(12),
reroute_sales_rep empidNOT NULL,
reroute_user sysname DEFAULT SUSER_SNAME( ),
reroute_changed datetime DEFAULT GETDATE( ) )
GO
CREATE TRIGGER update_person_data
ON people
FOR update AS
IF (COLUMNS_UPDATED(people_name)
OR COLUMNS_UPDATEEE(people_addr)
OR COLUMNS_UPDATED(city) )
BEGIN
-- Audit OLD record.
INSERT INTO people_reroute (reroute_log_type, reroute_people_id,
reroute_people_name, reroute_people_addr, reroute_city)
SELECT 'old', d.people_id, d.people_name, d.people_addr, d.city
FROM deleted AS d
-- Audit NEW record.
INSERT INTO people_reroute (reroute_log_type, reroute_people_id,
reroute_people_name, reroute_people_addr, reroute_city)
SELECT 'new', n.people_id, n.people_name, n.people_addr, n.city
FROM inserted AS n
END
GO
```

Note that SQL Server *CREATE* statements allow *deferred name resolution*, meaning that a command is processed even if it refers to a database object that does not yet exist in the database.

See Also

CREATE?ALTER FUNCTION/PROCEDURE
DELETE
DROP
INSERT
UPDATE

CREATE/ALTER TYPE Statement

The *CREATE TYPE* statement allows you to create a *user-defined type* (UDT); that is, a user-defined datatype or "class" in object-oriented terms. UDTs extend SQL capabilities into the realm of object-oriented programming by allowing inheritance and other object-oriented features. You can also create something called *typed tables* with the *CREATE TABLE* statement using a previously created type made with the *CREATE TYPE* statement. Typed tables are based on UDTs and are equivalent to "instantiated classes" from object-oriented programming.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with variations
SQL Server	Not supported

SQL2003 Syntax

```
CREATE TYPE type_name
[UNDER supertype_name]
[AS [new_udt_name] datatype [attribute] [...]]
  {[REFERENCES ARE [NOT] CHECKED
  [ON DELETE
  {[NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT] } ] |
  [DEFAULT value] |
  [COLLATE collation_name] } ]
  [ [NOT] INSTANTIABLE]
  [ [NOT] FINAL]
[REF IS SYTEM GENERATED |
  REF USING datatype
  [CAST {(SOURCE AS REF) | (REF AS SOURCE) WITH identifier} ] |
  REF new_udt_name [...] } ]
[CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH identifier]
[method_definition [...] ]
```

The syntax below alters an existing user-defined datatype:

```
ALTER TYPE type_name {ADD ATTRIBUTE type_definition |
  DROP ATTRIBUTE type_name}
```

Keywords

{CREATE | ALTER} TYPE *type_name*

Creates a new type or alters an existing type of *type_name*.

UNDER *supertype_name*

Creates a subtype that is dependent upon a single, pre-existing, named supertype. (A UDT can be a supertype if it is defined as *NOT FINAL*.)

AS [*new_udt_name*] *datatype* [*attribute*] [,...]

Defines attributes of the type as if they were column declarations in a *CREATE TABLE* statement without constraints. You should either define the UDT attribute on an existing datatype, like *VARCHAR(10)*, or on another, previously created UDT or even on a user-defined domain. Defining a UDT using a predefined datatype (e.g., *CREATE TYPE my_type AS INT*) creates a *distinct type*, while a UDT defined with an attribute definition is a *structured type*. The allowable attributes for a structured type are below:

ON DELETE NO ACTION

Produces an error when the foreign key is violated (the default).

ON DELETE RESTRICT

A synonym for *NO ACTION*.

ON DELETE CASCADE

Sets the value of the referencing column to the value of the referenced column.

ON DELETE SET NULL

Sets the referencing column value to NULL.

ON DELETE SET DEFAULT value

Defines a default value for the UDT when no value is supplied by the user. Follows the rules of a *DEFAULT* in the [CREATE/ALTER TABLE Statement](#) section.

COLLATE collation_name

Assigns a collation — that is, a sort order — for the UDT. When omitted, the collation of the database where the UDT was created applies. Follows the rules of a *COLLATION* in the [CREATE/ALTER TABLE Statement](#) section.

[*NOT*] *INSTANTIABLE*

Defines the UDT such that it can be instantiated. *INSTANTIABLE* is required for typed tables, but not standard UDTs.

[*NOT*] *FINAL*

Required for all UDTs. *FINAL* means the UDT may have no subtypes. *NOT FINAL* means the UDT may have subtypes.

REF

Defines either a system-generated or user-generated reference specification — that is, a sort of unique identifier that acts as a pointer that another type may reference. By referencing a pre-existing type using its reference specification, new types may inherit properties of pre-existing types. There are three ways to tell the DBMS how the typed table's reference column gets its values (i.e., its reference specification):

new_udt_name [,...]

Declares the reference specification is provided by another pre-existing UDT called *new_udt_name*.

IS SYSTEM GENERATED

Declares the reference specification is system-generated. (Think of an automatically incrementing column.) This is the default when the *REF* clause is omitted.

USING *datatype* [CAST {(SOURCE AS REF) | (REF AS SOURCE) WITH *identifier*]

Declares that the user defines the reference specification. You do this by using a predefined datatype and optionally casting the value. You can *CAST (SOURCE AS REF) WITH identifier* to cast the value with the specified *datatype* to the reference type of the structured type. You can *CAST (REF AS SOURCE) WITH identifier* to cast the value for the structured type to the *datatype*. The *WITH* clause allows you to declare an additional identifier for the cast datatype.

CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH *identifier method_definition* [,...]

Defines one or more pre-existing methods for the UDT. A *method* is merely a specialized user-defined function and is created using the *CREATE METHOD* statement (see *CREATE FUNCTION*). The *method_definition* clause is not needed for structured types since their method(s) are implicitly created. The default characteristics of a method are *LANGUAGE SQL*, *PARAMETER STYLE SQL*, *NOT DETERMINISTIC*, *CONTAINS SQL*, and *RETURN NULL ON NULL INPUT*.

ADD ATTRIBUTE *type_definition*

Adds an additional attribute to an existing UDT, using the format described earlier under the *AS* clause. Available via the *ALTER TYPE* statement.

DROP ATTRIBUTE *type_name*

Drops an attribute from an existing UDT. Available via the *ALTER TYPE* statement.

Rules at a Glance

You are able to create user-defined types (UDTs) as a way to further ensure data integrity in your database and to ease the work involved in doing so. An important concept of UDTs is that they allow you to easily create *subtypes*. Subtypes are UDTs built upon other UDTs (the supertype). Subtypes inherit the characteristics of their supertypes. The UDT that subtypes are dependent on is called a parent type or *supertype*.

Assume, for example, that you want to define a general UDT for phone numbers called **phone_nbr**. You could then easily build new subtypes of *phone_nbr* called **home_phone**, **work_phone**, **cell_phone**, **pager_phone**, etc. Each of the subtypes could then inherit the general characteristics of the parent type, but then have characteristics of their own. In the following example, we create a general root UDT called **money** and then several subtypes:

```
CREATE TYPE money (phone_number DECIMAL (10,2) )
```

```
    NOT FINAL;
```

```
CREATE TYPE dollar UNDER money AS DECIMAL(10,2)
```

```
    (conversion_rate DECIMAL(10,2) ) NOT FINAL;
```

```
CREATE TYPE euro  UNDER money AS DECIMAL(10,2)
```

```
    (dollar_conversion_rate DECIMAL(10,2) ) NOT FINAL;
```

```
CREATE TYPE pound UNDER euro
```

```
(euro_conversion_rate DECIMAL(10,2) ) FINAL;
```

Programming Tips and Gotchas

The biggest programming gotcha for user-defined types is that they are simply seldom used or well understood by most database developers and database administrators. Consequently, they can be problematic due to simple ignorance. They offer, however, a consistent and labor-saving approach for representing commonly reused conceptual elements in a database, such as address (e.g., **street1**, **street2**, **city**, **state**, **postal code**).

DB2

DB2 supports most elements of the ANSI standard with a few specific enhancements. The *CREATE TYPE* syntax is:

```
CREATE [DISTINCT] TYPE type_name  
[UNDER supertype_name]  
[AS [new_udt_name] datatype  
[WITH COMPARISONS]  
[[NOT] INSTANTIABLE] [INLINE LENGTH int] [WITHOUT COMPARISONS]  
[NOT FINAL] MODE DB2SQL  
[REF USING datatype] [CAST {(SOURCE AS REF) | (REF AS SOURCE)}  
WITH identifier } ]  
[[OVERRIDING] METHOD method_definition [,... ]
```

The syntax below alters an existing user-defined datatype on DB2:

```
ALTER TYPE type_name {ADD ATTRIBUTE type_definition |  
DROP ATTRIBUTE type_name [RESTRICT] |  
ADD METHOD method_definition |  
ALTER METHOD method_name [[NOT] FENCED] [[NOT] THREADSAFE] |  
DROP METHOD method_name [RESTRICT] }
```

The parameters are:

DISTINCT

Defines a distinct type. A distinct type is always sourced from one of the existing datatypes that ships with DB2. *CREATE DISTINCT TYPE* generates one or more function to cast between the distinct type and its source type.

WITH COMPARISONS

Generates support for the comparison operators (=, <>, <, <=, >, and >=) for use with a distinct type. This clause is used only with *CREATE DISTINCT TYPE*. Even when creating a distinct type, *WITH COMPARISONS* should not be used if the source *datatype* is *BLOB*, *CLOB*, *DBCLOB*, *LONG VARCHAR*, *LONG VARGRAPHIC*, or *DATALINK*.

INLINE LENGTH *int*

Defines the maximum length in bytes, *int*, of a structured type column. When omitted, DB2 calculates the maximum length itself.

WITHOUT COMPARISONS

Declares that there are no comparison functions supported for the structure type.

MODE DB2SQL

This is a required clause needed to maintain ANSI compliance.

OVERRIDING

Declares that the method of a subtype overrides a method of the same name on the supertype. This allows easy reimplementations of methods in subtypes.

METHOD *method_definition* [...]

DB2 supports a number of SQL routine characteristics and external routine characteristics on the *method_definition*. In short, you must define a method in the *CREATE/ALTER TYPE* statement, but then you must also define the implementation of the method using *CREATE METHOD*. Creating methods are explained in [CREATE/ALTER FUNCTION/PROCEDURE Statements](#).

ADD METHOD *method_definition*

Adds a new method, *method_definition*, to the existing type, *type_name*. However, the method is not usable until a *CREATE METHOD* statement defines the implementation of the method.

ALTER METHOD *method_name*

Alters a method, but the method must not be *LANGUAGE SQL*.

[NOT] *FENCED*

Declares whether the method is safe to run in the database's operating system process (*NOT FENCED*) or not (*FENCED*). *NOT FENCED* methods can introduce integrity problems when not adequately set up to prevent errors. This clause is incompatible with *NOT THREADSAFE*.

[NOT] *THREADSAFE*

Declares whether the method is safe to run in the same process as other routines (*THREADSAFE*) or not (*NOT THREADSAFE*).

DROP METHOD *method_name*

Drops an existing method of *method_name*.

RESTRICT

Disallows an attribute or method from being dropped when the type, *type_name*, is used in other existing objects, such as a table, view, column, index extension, etc.

In this example, we create a supertype for **employee** and a subtype for **manager**:

```
CREATE TYPE employee AS
```

```
  (last_name VARCHAR(32),
```

```
   emp_id INT,
```

```
   dept REF(dept),
```

```
   salary DECIMAL(10,2))
```



```
MODE DB2SQL;  
  
CREATE TYPE manager UNDER emp AS  
    (bonus DECIMAL(10,2))  
  
MODE DB2SQL;
```

The following example shows a more elaborate type hierarchy for addresses:

```
CREATE TYPE address_type AS  
    (street1  VARCHAR(30),  
     street2  VARCHAR(30),  
     city     VARCHAR(30),  
     state    VARCHAR(10),  
     locality_code CHAR(15) )  
  
NOT FINAL  
  
MODE DB2SQL  
  
    METHOD SAMEZIP (addr address_type)  
  
    RETURNS INTEGER  
  
    LANGUAGE SQL  
  
    DETERMINISTIC  
  
    CONTAINS SQL  
  
    NO EXTERNAL ACTION  
  
    METHOD DISTANCE (address_type)  
  
    RETURNS FLOAT  
  
    LANGUAGE C  
  
    DETERMINISTIC  
  
    PARAMETER STYLE SQL  
  
    NO SQL  
  
    NO EXTERNAL ACTION  
  
CREATE TYPE spain_address_type UNDER address_type AS  
    (family_name VARCHAR(30))  
  
NOT FINAL  
  
MODE DB2SQL  
  
CREATE TYPE usa_address_type UNDER address_type AS  
    (zip VARCHAR(10))  
  
NOT FINAL  
  
MODE DB2SQL
```

In the above example, we created a type called **address_type** with two subtypes: **spain_address_type** and **usa_address_type**.

The example that follows shows how to create a distinct type in DB2 used to identify an employee mailstop.

```
CREATE DISTINCT TYPE mailstop AS CHAR(5)

WITH COMPARISONS;
```

In essence, the type **mailstop** now functions like a regular datatype and may be used to create columns in tables, variables in stored procedures, etc. The *WITH COMPARISONS* clause tells DB2 to create the standard pre-existing system operators for use with the distinct type. Specific functions, such as *AVG*, *MIN*, or *LENGTH*, may not work unless the *CREATE FUNCTION* statement is used to register the function for the distinct type.

MySQL

Not supported.

Oracle

Oracle has *CREATE TYPE* and *ALTER TYPE* statements, but they are not ANSI standard. Instead of a single *CREATE TYPE* statement, Oracle uses *CREATE TYPE BODY* to define the code that makes up the UDT, while *CREATE TYPE* defines the argument specification for the type. The syntax for *CREATE TYPE* is:

```
CREATE [OR REPLACE] TYPE type_name

[AUTHID {DEFINER | CURRENT_USER} ]

{ [ UNDER supertype_name ] |

  {AS | IS} [OID 'object_identifier' ]

  {OBJECT | TABLE OF datatype | VARRAY ( limit ) OF datatype} ] }

  EXTERNAL NAME java_ext_name LANGUAGE JAVA USING data_definition

  { [ ( attribute datatype [,...] [EXTERNAL NAME 'name' ] |

    [[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]

    [{ {MEMBER | STATIC}

      {function_based | procedure_based} | constructor_clause |

      map_clause } [...] ]

    [pragma_clause] ] }
```

Once the type has been declared, you encapsulate all of the UDT logic in the type body declaration. The *type_name* for both the *CREATE TYPE* and *CREATE TYPE BODY* should be identical. The syntax for the type body is shown below:

```
CREATE [OR REPLACE] TYPE BODY type_name

{AS | IS}

{{MEMBER | STATIC}

{function_based | procedure_based | constructor_clause}}[...]

[map_clause]
```

Oracle's implementation of *ALTER TYPE* enables you to drop or add attributes and methods from the type:

```
ALTER TYPE type_name

{COMPILE [DEBUG] [{SPECIFICATION | BODY}]

  [compiler_directives] [REUSE SETTINGS] |

  REPLACE [AUTHID {DEFINER | CURRENT_USER}] AS OBJECT

  ( attribute datatype [,...] [element_definition [,...] ] ) |
```

```
[[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]
{ {ADD | DROP} { {MAP | ORDER} MEMBER FUNCTION function_name
  (parameter datatype [,...]) } |
  {MEMBER | STATIC} {function_based | procedure_based} |
  constructor_clause | map_clause } [...] |
  {ADD | DROP | MODIFY} ATTRIBUTE (attribute [datatype] [,...]) |
  MODIFY {LIMIT int | ELEMENT TYPE datatype} }
[ {INVALIDATE |
  CASCADE [{ [NOT] INCLUDING TABLE DATA | CONVERT TO SUBSTITUTABLE}]
  [FORCE] [EXCEPTIONS INTO table_name] } ]
```

Parameters for the three statements are as follows:

OR REPLACE

Recreates the UDT if it already exists. Objects that depend on the type are marked as *DISABLE* after you recreate the type.

AUTHID {DEFINER | CURRENT_USER}

Determines what user permissions any member functions or procedures are executed under and how external name references are resolved. (Note that subtypes inherit the permission style of their supertype.) This clause can only be used with an *OBJECT* type, not with a *VARRAY* type or nested table type, according to the following keywords:

DEFINER

Executes functions or procedures under the privileges of the user who created the UDT. Also specifies that unqualified object names (object names without a schema definition) in SQL statements are resolved to the schema where the member functions or procedures reside.

CURRENT_USER

Executes functions or procedures under the privileges of the user who invoked the UDT. Also specifies that unqualified object names in SQL statements are resolved to the schema of the user who invoked the UDT.

UNDER *supertype_name*

Declares that the UDT is a subtype of another, pre-existing UDT. The supertype UDT must be created with the *AS OBJECT* clause. A subtype will inherit the properties of the supertype, though you should override or add new properties to differentiate it from the supertype.

OID '*object_identifier*'

Declares an equivalent identical object, of name '*object_identifier*', in more than one database. This clause is most commonly used by those developing Oracle Data Cartridges and is seldom used in standard SQL statement development.

AS OBJECT

Creates the UDT as a root object type, the top level object in a UDT hierarchy of objects.

AS TABLE OF *datatype*

Creates a named nested table type of a UDT called *datatype*. The datatype cannot be an *NCLOB*, but *CLOB* or *BLOB* are acceptable. If the datatype is an object type, then the columns of the nested table must match the name and attributes of the object type.

AS VARRAY (*limit*) OF *datatype*

Creates the UDT as an ordered set of elements, all of the same datatype. The *limit* is an integer of zero or more. The type name must be a built-in datatype, a *REF*, or an object type. The *VARRAY* cannot contain *LOB* or *XMLType* datatypes.

EXTERNAL NAME *java_ext_name* LANGUAGE JAVA USING *data_definition*

Maps a Java class to a SQL UDT by specifying the name of a public Java external class, *java_ext_name*. Once defined, all objects in a Java class must be Java objects. The *data_definition* may be *SQLData*, *CustomDatum*, or *OraData* as defined in the "Oracle9i JDBC Developers Guide." You can map many Java object types to the same class, but there are two restrictions. First, do not map to the same class two or more subtypes of a common datatype. Second, subtypes must be mapped to an immediate subclass of the class that their supertype is mapped to.

datatype

Declares the attributes and datatypes used by the UDT. Oracle does not allow *ROWID*, *LONG*, *LONG ROW*, or *UROWID*. Nested tables and varrays do not allow attributes of **AnyType**, **AnyData**, or **AnyDataSet**.

EXTERNAL NAME '*name*' [NOT] OVERRIDING

Declares that this method overrides a *MEMBER* method defined in the supertype (*OVERRIDING*) or not (*NOT OVERRIDING*), the default. This clause is valid only for *MEMBER* clauses.

MEMBER | *STATIC*

Describes the way in which subprograms are associated with the UDT as attributes. *MEMBER* has an implicit first argument referenced as *SELF*, as in *object_expression.method()*. *STATIC* has no implicit arguments, as in *type_name.method()*. The following basis are allowed to call subprograms:

function_based

Declares a subprogram that is function-based using the syntax:

```
FUNCTION function_name ( parameter datatype [...] )
```

```
return_clause | java_object_clause
```

The *function_name* cannot be the name of any existing attribute, include those inherited from supertypes. This clause allows you to define the PL/SQL function-based UDT body without resorting to the *CREATE TYPE BODY* statement. The *return_clause* and *java_object_clause* are defined in the following text.

procedure_based

Declares a subprogram that is function-based using the syntax:

```
PROCEDURE procedure_name ( parameter datatype [...] )
```

```
{AS | IS} LANGUAGE {java_call_spec | c_call_spec}
```

Refer to the entry on *java* and *c_call_spec* later in this list. This clause allows you to define the PL/SQL procedure-based UDT body without resorting to the *CREATE TYPE BODY* statement. The *procedure_name* cannot be the name of any existing attribute, include those inherited from supertypes.

constructor_clause

Declares one or more constructor specifications using the syntax:

```
[FINAL] [INSTANTIABLE] CONSTRUCTOR FUNCTION datatype  
  [ ( [SELF IN OUT datatype , ] {parameter datatype [...] } ) ]  
RETURN SELF AS RESULT  
  [ {AS | IS} LANGUAGE {java_call_spec | c_call_spec}
```

A constructor specification is a function that returns an initialized instance of a UDT. Constructor specifications are always *FINAL*, *INSTANTIABLE*, and *SELF IN OUT*, so these keywords are not required. The *java_call_spec* and *c_call_spec* subclauses may be replaced with a PL/SQL code block in the *CREATE TYPE BODY* statement. (Refer to the entries on *java_call_spec* and *c_call_spec* later in this list.)

map_clause

Declares the mapping or ordering of a supertype using the syntax:

```
{MAP | ORDER} MEMBER function_based
```

Refer to the entry on *function_based* earlier in this list. *MAP* uses more efficient algorithms for object comparison and is best in situations when performing extensive sorting or hash joins. *MAP MEMBER* specifies the relative position of a given instance in the ordering of all instances of the UDT. *ORDER MEMBER* specifies the explicit position of an instance and references a *function_based* subprogram that returns a NUMBER datatype value.

return_clause

Declares the datatype return format of the SQL UDT using the syntax:

```
RETURN datatype [ {AS | IS} LANGUAGE {java_call_spec |  
  c_call_spec}.  
.
```

java_object_clause

Declares the return format of the Java UDT using the syntax:

```
RETURN {datatype | SELF AS RESULT} EXTERNAL [VARIABLE] NAME  
  'java_name'
```

If you use the *EXTERNAL* clause, the value of the public Java method must be compatible with the SQL returned value.

pragma_clause

Declares a *pragma restriction* (that is, an Oracle precompiler directive) for the type using the syntax:

```
PRAGMA RESTRICT REFERENCES ( {DEFAULT | method_name},  
{RNDS | WNDS |RNPS | WNPS | TRUST}{,...} )
```

This feature is deprecated and *should be avoided*. It is intended to control how UDTs read and write database tables and variables. The parameters in the *PRAGMA* syntax are:

DEFAULT

Applies the pragma to all methods in the type that don't have another pragma in place.

method_name

Identifies the exact method to which to apply the pragma.

RNDS

Reads no database state — no database reads allowed.

WNDS

Writes no database state — no database writes allowed.

RNPS

Reads no package state — no package reads allowed.

WNPS

Writes no package state — no package writes allowed.

TRUST

States that the restrictions of the pragma are assumed but not enforced.

java_call_spec

Identifies the Java implementation of a method using the syntax *JAVA NAME 'string'*. This clause allows you to define the Java UDT body without resorting to the *CREATE TYPE BODY* statement.

c_call_spec

Declares a C language call specification using the syntax:

C [*NAME name*] *LIBRARY lib_name* [*AGENT IN (argument)*]
[*WITH CONTEXT*] [*PARAMETERS (parameter [,...])*]

This clause allows you to define the C UDT body without resorting to the *CREATE TYPE BODY* statement.

COMPILE

Compiles the object type specification and body. This is the default when neither a *SPECIFICATION* clause nor a *BODY* clause are defined.

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive = 'value'*. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

DEBUG

Generates and stores additional codes for the PL/SQL debugger. Do not specify both *DEBUG* and the *compiler_directive PLSQL_DEBUG*.

SPECIFICATION | BODY

Chooses whether to recompile the *SPECIFICATION* of the object type (created by the *CREATE TYPE* statement) or the *BODY* (created by the *CREATE TYPE BODY* statement).

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive* = '*value*'. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

REUSE SETTINGS

Retains the original value for the *compiler_directives* mentioned earlier.

REPLACE AS OBJECT

Adds new member subtypes to the specification. This clause is valid only for object types.

[NOT] OVERRIDING

Indicates the method overrides a *MEMBER* method defined in the supertype. This clause is only valid with *MEMBER* methods and is required for methods that define (or redefine using *ALTER*) a supertype method. *NOT OVERRIDING* is the default if this clause is omitted.

ADD

Adds a new *MEMBER* function, function- or procedure-based subprogram, or attribute to the UDT.

DROP

Drops an existing *MEMBER* function, function- or procedure-based subprogram, or attribute from the UDT.

MODIFY

Alters the properties of an existing attribute of the UDT.

MODIFY LIMIT int

Increases the number of elements in a *VARRAY* collection type up to *int*, as long as *int* is greater than the current number of elements in the *VARRAY*. It is not valid for nested tables.

MODIFY ELEMENT TYPE datatype

Increases the precision, size, or length of a scalar datatype of a *VARRAY* or nested table. This clause is valid for any non-object collection type. If the collection is a *NUMBER*, you may increase its precision or scale. If the collection is *RAW*, you may increase its maximum size. If the collection is *VARCHAR2* or *NVARCHAR2*, you may increase its maximum length.

INVALIDATE

Invalidates all dependent objects without checks.

CASCADE

Cascades the change to all subtypes and tables. By default, the action will be rolled back if any errors are encountered in the dependent types or tables.

[NOT] INCLUDING TABLE DATA

Converts data stored in the UDT columns to the most recent version of the column's type (*INCLUDING TABLE DATA*), the default, or not (*NOT INCLUDING TABLE DATA*). When *NOT*, Oracle checks the metadata, but does not check or update the dependent table data.

CONVERT TO SUBSTITUTABLE

Used when changing a type from *FINAL* to *NOT FINAL*. The altered type then can be used in substitutable tables and columns, as well as in subtypes, instances of dependent tables, and columns.

FORCE

Causes the *CASCADE* operation to go forward, ignoring any errors found in dependent subtypes and tables. All errors are logged to a previously created *EXCEPTIONS* table.

EXCEPTIONS INTO *table_name*

Logs all error data to a table previously created using the system package called `DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE`.

In this example, we create a Java SQLJ object type called **address_type**:

```
CREATE TYPE address_type AS OBJECT
  EXTERNAL NAME 'scott.address' LANGUAGE JAVA
  USING SQLDATA ( street1 VARCHAR(30) EXTERNAL NAME 'str1',
    street2      VARCHAR(30) EXTERNAL NAME 'str2',
    city         VARCHAR(30) EXTERNAL NAME 'city',
    state        CHAR(2)   EXTERNAL NAME 'st',
    locality_code CHAR(15) EXTERNAL NAME 'lc',
    STATIC FUNCTION square_feet RETURN NUMBER
      EXTERNAL VARIABLE NAME 'square_feet',
    STATIC FUNCTION create_addr (str VARCHAR,
      City VARCHAR, state VARCHAR, zip NUMBER)
      RETURN address_type
      EXTERNAL NAME 'create (java.lang.String,
        java.lang.String, java.lang.String, int)
        return scott.address',
    MEMBER FUNCTION rtrim_spaces RETURN SELF AS RESULT
      EXTERNAL NAME 'rtrim_spaces ( ) return scott.address' )
NOT FINAL;
```

We could create a UDT using a *VARRAY* type with four elements:

```
CREATE TYPE employee_phone_numbers AS VARRAY(4) OF CHAR(14);
```

In the following example, we alter the **address_type** that we created earlier by adding a *VARRAY* called **phone_varray**:

```
ALTER TYPE address_type
  ADD ATTRIBUTE (phone phone_varray) CASCADE;
```


In this last example, we'll create a supertype and subtype called **menu_item_type** and **entry_type**, respectively:

```
CREATE OR REPLACE TYPE menu_item_type AS OBJECT
(id INTEGER, title VARCHAR2(500),
  NOT INSTANTIABLE
  MEMBER FUNCTION fresh_today
  RETURN BOOLEAN)
NOT INSTANTIABLE
NOT FINAL;
```

In the above example, we created a type specification (but not the type body) that defines items that may appear on a menu at a café. Included with the type specification is a subprogram method called **fresh_today**, a Boolean indicator that tells whether the menu item is made fresh that day. The *NOT FINAL* clause that appears at the end of the code tells Oracle that this type may serve as the supertype (or base type) to other subtypes that we might derive from it. So now, let's create the **entre_type**:

```
CREATE OR REPLACE TYPE entry_type UNDER menu_item_type
(entre_id INTEGER, desc_of_entre VARCHAR2(500),
  OVERRIDING MEMBER FUNCTION fresh_today
  RETURN BOOLEAN)
NOT FINAL;
```

PostgreSQL

PostgreSQL supports a *CREATE TYPE* statement used to create a new datatype, but does not support an *ALTER TYPE* statement. PostgreSQL's implementation of the *CREATE TYPE* statement is nonstandard:

```
CREATE TYPE type_name
( INPUT = input_function_name,
  OUTPUT = output_function_name,
  INTERNALLENGTH = { int | VARIABLE }
[, DEFAULT = value]
[, ELEMENT = array_element_datatype]
[, DELIMITER = delimiter_character]
[, PASSEDBYVALUE ]
[, ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE} ]
[, STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN} ] )
```

The parameters are as follows:

```
CREATE TYPE type_name
```

Creates a new user-defined datatype called *type_name*. The name may not exceed 30 characters in length nor begin with an underscore.

INPUT = *input_function_name*

Declares the name of a previously created function that converts external argument values into a form usable by the type's internal form.

OUTPUT = *output_function_name*

Declares the name of a previously created function that converts internal output values to a display format.

INTERNALLENGTH = { *int* | VARIABLE }

Specifies a numeric value, *int*, for the internal length of the new type, if the datatype is fixed length. The keyword *VARIABLE* declares that the internal length is variable. The default is *VARIABLE*.

DEFAULT = *value*

Defines a value for *type* when it defaults.

ELEMENT = *array_element_datatype*

Declares that the datatype is an array and that *array_element_datatype* is the datatype of the array elements. For example, an array containing integer values would be *ELEMENT = INT4*. In general, you should allow the *array_element_datatype* value to default. The only time you might want to override the default is when creating a fixed-length UDT composed of an array of multiple identical elements that need to be directly accessible by subscripting.

DELIMITER = *delimiter_character*

Declares a character used as a delimiter between output values of an array produced by the type. Only used with the *ELEMENT* clause. The default is a comma.

PASSEDBYVALUE

Specifies that the values of the datatype are passed by value and not by reference. The optional clause cannot be used for types whose value is longer than the *DATUM* datatype (4 bytes on most operating systems, 8 bytes on a few others).

ALIGNMENT = { *CHAR* | *INT2* | *INT4* | *DOUBLE* }

Defines a storage alignment for the type. Four datatypes are allowed, with each one equating to a specific boundary. *CHAR* equals a 1-byte boundary. *INT2* equals a 2-byte boundary. *INT4* equals a 4-byte boundary (the requirement for a variable-length UDT on PostgreSQL). *DOUBLE* equals an 8-byte boundary.

STORAGE = { *PLAIN* | *EXTERNAL* | *EXTENDED* | *MAIN* }

Defines a storage technique for variable-length UDTs. (*PLAIN* is required for fixed-length UDTs.) Four types are allowed:

PLAIN

Stores the UDT in-line, when declared as the datatype of a column in a table, and uncompressed.

EXTERNAL

Stores the UDT outside of the table without trying to compress it first.

EXTENDED

Stores the UDT as a compressed value if it fits inside of the table. If it is still too long, PostgreSQL will save the UDT outside of the table.

MAIN

Stores the UDT as a compressed value within the table. There are situations where PostgreSQL cannot save the UDT within the table because it is just too large. The *MAIN* storage parameter puts the highest emphasis on storing UDTs with all other table data.

When you create a new datatype in PostgreSQL, it is available *only* in the current database. The user who created the datatype is the owner.

In order to create a new datatype, you must create at least two functions before defining the type (see [CREATE/ALTER FUNCTION/PROCEDURE Statements](#)). In summary, you must create an *INPUT* function that provides the type with external values that can be used by the operators and functions defined for the type. You must also create an *OUTPUT* function that renders a usable external representation of the datatype. There are some additional requirements when creating the input and output functions:

- The input function should either take one argument of type OPAQUE or take three arguments of OPAQUE, OID, and INT4. In the latter case, OPAQUE is the input text of a C string, OID is the element type for array types, and the INT4 (if known) is the typemod of the destination column.
- The output function should either take one argument of type OPAQUE or take two arguments of OPAQUE and OID. In the latter case, OPAQUE is the datatype itself and the OID is the element type for array types, if needed.

For example, we can create a UDT called **floorplan** and use it to define a column in two tables, one called **house** and one called **condo**:

```
CREATE TYPE floorplan
```

```
(INTERNALLENGTH=12, INPUT=squarefoot_calc_proc,
```

```
OUTPUT=out_floorplan_proc);
```

```
CREATE TABLE house
```

```
(house_plan_id int4,
```

```
size floorplan,
```

```
descrip varchar(30) );
```

```
CREATE TABLE condo
```

```
(condo_plan_id INT4,
```

```
size floorplan,
```

```
descrip varchar(30)
```

```
location_id varchar(7) );
```

SQL Server

Not supported. New datatypes can be added to SQL Server using the non-ANSI system stored procedure **sp_addtype**.

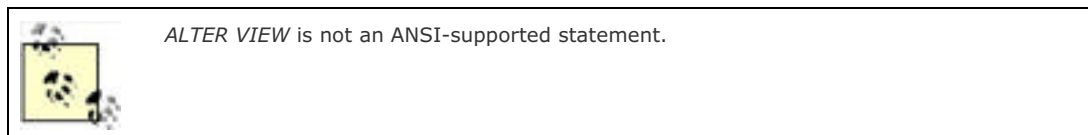
See Also

```
CREATE/ALTER FUNCTION/PROCEDURE  
DROP
```

CREATE/ALTER VIEW Statement

This statement creates a *view*, also known as a *virtual table*. A view acts just like a table but is actually defined as a query. When a view is referenced in a statement, the result set of the query becomes the content of the view for the duration of that statement. Almost any valid *SELECT* statement can define the contents of a view, though some platforms restrict the certain clauses of the *SELECT* statement and certain set operators.

In some cases, views can be updated, causing the view changes to be translated to the underlying data in the base tables. Some database platforms support a *materialized view*; that is, a physically created table that is defined with a query just like a view.



Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE [RECURSIVE] VIEW view_name {[column [,...]]} |  
[OF udt_name [UNDER supertype_name  
  [REF IS column_name {SYSTEM GENERATED | USER GENERATED | DERIVED} ]  
  [column_name WITH OPTIONS SCOPE table_name] ] }  
AS  
select_statement [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Keywords

CREATE VIEW *view_name*

Creates a new view using the supplied name.

RECURSIVE

Creates a view that derives values from itself. It must have a *column* clause and may not use the *WITH* clause.

[(*column* [,...])]

Names all of the columns in the view. The number of columns declared here must match the number of columns generated by the *select_statement*. When omitted, the columns assume the names of the columns that they come from. This clause is required when one or more of the columns is derived and does not have a base table column to reference.

OF *udt_name* [UNDER *supertype_name*]

Defines the view on a UDT rather than on the *column* clause. The typed view is created using each attribute of the type as a column in the view. Use the *UNDER* clause to define a view on a subtype.

REF IS *column_name* {SYSTEM GENERATED | USER GENERATED | DERIVED}

Defines the object-ID column for the view.

column_name WITH OPTIONS SCOPE *table_name*

Provides scoping for a reference column in the view. (Since the columns are derived from the type, there is no column list. Therefore, to specify column options, you must use "*column_name* WITH OPTIONS...".)

AS *select_statement*

Defines the exact *SELECT* statement that provides the data of the view.

[WITH [CASCADED | LOCAL] CHECK OPTION]

Used *only* on views that allow updates to their base table. It ensures that only data that may be read by the view may be inserted, updated, or deleted by the view. For example, if a view of **employees** showed salaried employees but not hourly employees, it would be impossible to insert, update, or delete hourly employees through that view. The *CASCADED* and *LOCAL* options of the *CHECK OPTION* clause are used for nested views. *CASCADED* performs the check option for the current view and all views it is built on top of. *LOCAL* performs the check option only for the current view, even when it is built upon other views.

Rules at a Glance

Views are usually as effective as the queries upon which they are based. That is why it is important to be sure that the defining *SELECT* statement is speedy and well-written. The simplest view is based on the entire contents of a single table:

```
CREATE VIEW employees
```

```
AS
```

```
SELECT *
```

```
FROM employee_tbl;
```

A column list also may be specified after the view name. The optional column list contains aliases serving as names for each element in the result set of the *SELECT* statement. If you use a column list, then you must provide a name for every column returned by the *SELECT* statement. If you don't use a column list, then the columns of the view will be named whatever the columns in the *SELECT* statement are called. You will sometimes see complex *SELECT* statements within a view that make heavy use of *AS* clauses for all columns, because that allows the developer of the view to put meaningful names on the columns without including a column list.

The ANSI standard defines that you must use a column list or an *AS* clause. However, some vendors allow more flexibility, so follow these rules for when to use an *AS* clause:

- When the *SELECT* statement contains calculated columns, such as (*salary * 1.04*)
- When the *SELECT* statement contains fully qualified column names, such as **pubs.scott.employee**
- When the *SELECT* statement contains more than one column of the same name (though with separate schema or database prefixes)

For example, the following two view declarations have the same functional result:

```
-- using a column-list
```

```
CREATE VIEW title_and_authors
```

```
(title, author_order, author, price, avg_monthly_sales,
```

```
publisher)
AS
SELECT t.title, ta.au_ord, a.au_lname, t.price, (t.ytd_sales / 12),
       t.pub_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;
-- using the AS clause with each column
CREATE VIEW title_and_authors
AS
SELECT t.title AS title, ta.au_ord AS author_order,
       a.au_lname AS author, t.price AS price,
       (t.ytd_sales / 12) AS avg_monthly_sales, t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0
```

You can alternately change the title of columns using the column list. In this case, we'll change **avg_monthly_sales** to **avg_sales**. Note that the code overrides the default column names provided by the AS clauses (in bold):

```
CREATE VIEW title_and_authors
(title, author_order, author, price, avg_sales, publisher)
AS
SELECT t.title AS title, ta.au_ord AS author_order,
       a.au_lname AS author, t.price AS price,
       (t.ytd_sales / 12) AS avg_monthly_sales, t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;
```

ANSI standard view can update the base table(s) it is based upon if it meets the following conditions:

- The view does not have *UNION*, *EXCEPT*, or *INTERSECT* operators.
- The defining *SELECT* statement does not contain *GROUP BY* or *HAVING* clauses.
- The defining *SELECT* statement does not contain any reference to non-ANSI pseudo-columns such as *ROWNUM* or *ROWGUIDCOL*.
- The defining *SELECT* statement does not contain the *DISTINCT* clause.

- The view is not materialized.

This example shows a view named **california_authors** that allows data modifications to apply only to authors within the state of California:

```
CREATE VIEW california_authors
AS
SELECT au_lname, au_fname, city, state
FROM authors
WHERE state = 'CA'
WITH LOCAL CHECK OPTION
```

The view shown in this example would accept *INSERT*, *DELETE*, and *UPDATE* statements against the base table, but still maintain that all inserted, updated, or deleted records contain a **state** of 'CA' using the *WITH ... CHECK* clause.

The most important rule to remember when updating a base table through a view is that all columns in a table that are defined as *NOT NULL* must receive a not null value when receiving a new or changed value. You can do this explicitly by directly inserting or updating a not null value into the column or by relying on a default value. In addition, views do not lift constraints on the base table. Thus, the values being inserted or updated into the base table must meet all the constraints placed on it by unique indexes, primary keys, *CHECK* constraints, etc.

Programming Tips and Gotchas

Views also can be built upon other views, but this is inadvisable and usually considered bad practice. Depending on the platform, such a view may take longer to compile, but may offer the same performance as a transaction against the base table(s). On other platforms, where each view is dynamically created as it is invoked, nested views may take a long time to return a result set because each level of nesting means another query must be processed before a result set is returned to the user. So in this worst case scenario, a three-level nested view must make three correlated query calls before it can return results to the user.

Although materialized views are defined like views, they take up space more like tables. Ensure that you have enough space available for the creation of materialized views.

DB2

DB2 supports a number of extensions to the ANSI standard *CREATE VIEW* statement:

```
CREATE VIEW view_name
{ [ ( column [,...] ) ] | [OF type_name MODE DB2SQL
    { (REF IS objectid_column_name USER GENERATED
        [UNCHECKED] [, with_options] ) |
        UNDER parent_view_name INHERIT SELECT PRIVILEGES
        [ ( with_options ) ] }
AS
[WITH common_table_expression [, ...] ]
(select_statement)
[WITH {CASCADE | LOCAL} CHECK OPTION]
```

The *ALTER VIEW* syntax is:

```
ALTER VIEW view_name
ALTER [COLUMN] column_name ADD SCOPE type_name
```

Parameters for both statements are as follows:

OF *type_name* MODE DB2SQL

Declares that the columns of the view are derived from attributes of a structured type called *type_name*. *MODE DB2SQL* indicates the type is DB2's version of a typed view.

REF IS *objectid_column_name* USER GENERATED

Declares that the object identifier column appears as the first column in the view. That first column in the *SELECT* clause of the view must be a *REF* type. In this case, the *objectID* is required on the topmost (i.e., root) view of a view hierarchy. The root view must be a typed view containing the *OF* clause that is not a subview. The *USER GENERATED* clause specifies that the first value of the *objectID* must be provided by the user. From then on, the value is maintained by the system.

UNCHECKED

Declares that the user is certain that all values in first column of the *SELECT* clause are unique and are not null. Uniqueness may be enforced by a primary key, unique key, unique index, *OID-column* definition, or an external method. This setting is usually optional, but is required for views in a typed hierarchy.

UNDER *parent_view_name* INHERIT SELECT PRIVILEGES

Declares that the view is a subview of the previously declared view *parent_view_name*. The child and parent views must be in the same schema. The parent view must have been declared with the *REF* clause and must be sourced from a structured type. The parent view named cannot have any subviews of the same structured type as the view being created. The *INHERIT SELECT PRIVILEGES* clause entitles any user or role with *SELECT* privileges on the parent view to also have *SELECT* privileges on all subviews.

with_options

Declares additional options for specific columns of the view. Multiple options are allowed by placing a comma between each option. The full syntax of the *with_options* clause is:

{*column_name* WITH OPTIONS {*READ ONLY* | *SCOPE type_name*} [,...]} [,...]

The parameters are:

READ ONLY

Specifies that a column is read-only.

SCOPE type_name

Specifies the scope of a reference type column where *type_name* is the name of the typed table or view. A scope is required for any column serving as the left operand of a dereference operator or as the argument of the *DEREF* function.

WITH *common_table_expression*

Associates a common table expression with the *SELECT* statement of the view. This clause is incompatible with typed views.

WITH {*CASCADE* | *LOCAL*} *CHECK OPTION*

Declares that any rows inserted or updated through the view must conform to the definition of the view. Any row that couldn't also be returned by the view cannot, therefore, be inserted or updated through it. When omitted, no checking of inserted or updated data is done beyond the constraints that exist on the source table. *WITH CHECK OPTION* is not allowed for read-only views, views with an *INSTEAD OF* trigger, or views that reference a nicknamed source, *NODENUMBER* or *PARTITION* functions, non-deterministic functions, or functions with external actions. The following *WITH CHECK OPTION* options are available:

CASCADE

Causes the search conditions of the current view to inherit the search conditions of any updateable view that the current view is based on. With this option, the current view has its own *WHERE* clause implicitly appended with an *AND* keyword to the *WHERE* clause of the view it depends on.

LOCAL

Applies the search conditions of the current view to any views that depend on it.

ALTER [COLUMN] *column_name* ADD SCOPE *type_name*

Allows you to add scope characteristics to an existing view.

The *SELECT* statement of a view should be a standard *SELECT* statement and cannot contain any of the following:

- Temporary tables
- Parameter markers
- Referenced host variables

You can get around the limitation on parameterized views by making a user-defined function based on a SQL table.

A typed view or subview must follow some specific rules:

- The *SELECT* statement of the view cannot reference *DBPARTITIONNUM* or *HASHEDVALUE* functions, functions with external actions, or any non-deterministic function.
- The view can contain a single *SELECT* statement. More *SELECT* statements can be added by using a *UNION ALL* clause between each one. (DB2 refers to a query with multiple *SELECT* statements as having multiple *branches*.)
- Branches should not cross hierarchies.
- Branches of a typed view should not contain *GROUP BY* or *HAVING* clauses.
- The root view should contain the *UNCHECKED* clause if it contains branches.

The following is an example including types, a typed table, and a view hierarchy, built around **people_type** and **emp_type**:

```
CREATE TYPE people_type
  AS (name VARCHAR(20), address VARCHAR(100))
  MODE DB2SQL
  CAST (SOURCE AS REF) WITH people_func;

CREATE TYPE emp_type
  UNDER people_type
  AS (title VARCHAR(20), salary DECIMAL(8,2))
  MODE DB2SQL
  CAST (SOURCE AS REF) WITH emp_func;

CREATE TABLE people
  OF people_type
  (REF IS people_oid USER GENERATED);

INSERT INTO people (people_oid, name, address)
```

```
VALUES (people_func('0'), 'Sammy', '123 Lane');

CREATE TABLE emp
  OF emp_type
  UNDER people INHERIT SELECT PRIVILEGES;

INSERT INTO emp (people_oid, name, address, title, salary)
  VALUES (emp_func('1'), 'Josh', '123 Street', 'Director', 1000);

CREATE VIEW people_view
  OF people_type
  MODE DB2SQL
  (REF IS people_view_oid USER GENERATED)
  AS SELECT people_func (VARCHAR(people_oid)), name, address
     FROM ONLY (people);

CREATE VIEW emp_view
  OF emp_type
  MODE DB2SQL
  UNDER people_view INHERIT SELECT PRIVILEGES
  AS SELECT emp_func (VARCHAR(people_oid)), name,
     address, title, salary
     FROM emp;

SELECT *
  FROM people_view;

PEOPLE_VIEW_OID NAME ADDRESS
-----
x'30'      Sammy 123 Lane
x'31'      Josh 123 Street

SELECT *
  FROM emp_view;

PEOPLE_VIEW_OID NAME ADDRESS TITLE SALARY
-----
x'31'      Josh 123 Street Director 1000.00
```

The *WITH CHECK OPTION* clause can be tricky because you are, in effect, telling the views to combine *WHERE* clauses either from upward in the dependency chain (via *CASCADE*) or downward in the dependency chain (via *LOCAL*). Consider the following:

```
V1 sourced from table T
V2 sourced from V1 WITH check_opn CHECK OPTION
V3 sourced from V2
V4 sourced from V3 WITH check_opn CHECK OPTION
V5 sourced from V4
```

The following table shows the search conditions against which inserted or updated rows are checked.



Note that if you create a view (or any other object, for that matter) with a schema name that does not already exist in the database, DB2 will implicitly create a schema of that name.

MySQL

Not supported.

Oracle

Oracle supports extensions to the ANSI standard to create object-oriented views, XMLType views, and views that support *LOB* and object types:

```
CREATE [OR REPLACE] [ [NO] FORCE] VIEW view_name
  {[(column [,...]) [constraint_clause] |
  [OF type_name {UNDER parent_view |
  WITH OBJECT IDENTIFIER {DEFAULT | (attribute [,...]) }
  [(constraint_clause)] ] |
  [OF XMLTYPE [ [XMLSCHEMA xml_schema_url] ELEMENT
  {element | xml_schema_url # element} ]
  WITH OBJECT IDENTIFIER {DEFAULT | (attribute [,...]) }
AS
(select statement)
[WITH [READ ONLY | CHECK OPTION [CONSTRAINT constraint_name] ] ]
```

Oracle's implementation of the *ALTER VIEW* statement supports added capabilities, such as adding, dropping, or modifying constraints associated with the view. In addition, the *ALTER VIEW* statement will explicitly recompile a view that is invalid, enabling you to locate recompile errors before runtime. This recompiling feature enables you to determine if a change in a base table negatively impacts any dependent views:

```
ALTER VIEW view_name
  {ADD constraint_clause |
  MODIFY CONSTRAINT constraint_clause [NO]RELY] |
  DROP {PRIMARY KEY | CONSTRAINT constraint_clause |
  UNIQUE (column [,...])} }
COMPILE
```

The parameters are:

OR REPLACE

Replaces any existing view of the same *view_name* with the new view.

[*NO*] *FORCE*

The *FORCE* clause creates the view regardless of whether the base tables exist or the user creating the view has privileges to the base tables, views, or functions defined in the view. The *FORCE* clause also creates the

view regardless of errors that occur during view creation. The *NO FORCE* clause creates the view only if the base tables and proper privileges are in place.

constraint_clause

Allows you to specify constraints (see *CREATE TABLE*) on views for *CREATE VIEW*. Using *ALTER VIEW*, this clause also allows you to affect a named, pre-existing constraint. You can define the constraint at the view level, similar to a table-level view, or at the column or attribute level. Note that although Oracle allows you to define constraints on a view, it doesn't yet enforce them. Oracle supports constraints on a view in *DISABLE* and *NOVALIDATE* modes.

OF *type_name*

Declares that the view is an object view of *type_name*. The columns of the view correspond directly to the attributes returned by *type_name*, where *type_name* is a previously declared type (see *CREATE TYPE*). You do not specify column names for object and XMLType views.

UNDER *parent_view*

Specifies a subview based on a pre-existing *parent_view*. The subview must be in the same schema as the parent view, the *type_name* must be an immediate subtype of the *parent_view*, and one subview is allowed.

WITH OBJECT IDENTIFIER {DEFAULT | (*attribute* [,...]) }

Defines the root object view as well as any *attribute* of the object type used to identify each row of the object view. The attributes usually correspond to the primary key columns of the base table and must uniquely identify each row of the view. This clause is incompatible with subviews and dereferenced or pinned *REF* keys. The *DEFAULT* keyword uses the implicit object identifier of the base object table or view.

OF XMLTYPE [[XMLSCHEMA *xml_schema_url*] ELEMENT { *element* | *xml_schema_url* # *element* } WITH OBJECT IDENTIFIER (*attribute* [,...])

Specifies that the view will return XMLType instances. Specifying the optional *xml_schema_url* as a pre-registered XMLSchema and element name further constrains the returned XML as an element in that XMLSchema. The *WITH OBJECT IDENTIFIER* clause specifies the identifier that uniquely identifies each row of the XMLType view. One or more attributes may use non-aggregate functions like **extractValue()** to obtain the identifiers from the resultant XMLType.

WITH READ ONLY

Ensures the view is used only to retrieve data and not to modify it.

WITH CHECK OPTION [CONSTRAINT *constraint_name*]

Forces the view to accept only inserted and updated data that can be returned by the view's *SELECT* statement. Alternately, you can specify a single *CHECK OPTION constraint_name* that exists on the base table that you want to enforce. If the constraint is not named, Oracle names the constraint *SYS_Cn*, where *n* is an integer.

ADD *constraint_clause*

Adds a new constraint to the view. Oracle supports constraints only in *DISABLE* and *NOVALIDATE* mode.

MODIFY CONSTRAINT *constraint_clause* [NO]RELY

Changes the *RELY* or *NORELY* setting of an existing view constraint. (*RELY* and *NORELY* are explained under *CREATE TABLE*.)

DROP {PRIMARY KEY | CONSTRAINT *constraint_clause* | UNIQUE (*column* [,...]) }

Drops an existing constraint on a view.

COMPILE

Recompiles the view.

Any dblink in the view's *SELECT* statement must be declared using the *CREATE DATABASE LINK...CONNECT TO* statement. Views containing flashback queries will have their *AS OF* clause evaluated at each invocation of the view, not when the view is compiled.

In this example, we create a view that has an added constraint:

```
CREATE VIEW california_authors (last_name, first_name,  
    author_ID UNIQU RELY DISABLE NOVALIDATE,  
    CONSTRAINT id_pk PRIMARY KEY (au_id) RELY DISABLE NOVALIDATE)  
AS  
SELECT au_lname, au_fname, au_id  
FROM authors  
WHERE state = 'CA';
```

We might also wish to create an object view on an Oracle database and schema. This example creates the type and the object view:

```
CREATE TYPE inventory_type AS OBJECT  
( title_id NUM(6),  
    warehouse wrhs_typ,  
    qty NUM(8) );  
CREATE VIEW inventories OF inventory_type  
WITH OBJECT IDENTIFIER (title_id)  
AS  
SELECT i.title_id, wrhs_typ(w.wrhs_id, w.wrhs_name,  
    w.location_id), i.qty  
FROM inventories i  
JOIN warehouses w ON i.wrhs_id = w.wrhs_id;
```

You could recompile the **inventory_type** view like this:

```
ALTER VIEW inventory_type COMPILE;
```

An updateable view in Oracle cannot contain any of the following:

- The *DISTINCT* clause
- *UNION*, *INTERSECT*, or *MINUS*
- Joins that cause inserted or updated data to affect more than one table
- Aggregate or analytic functions
- *GROUP BY*, *ORDER BY*, *CONNECT BY*, or *START WITH* clauses
- Subqueries or collection expressions in the *SELECT* item list (Subqueries are acceptable in the *SELECT* statement's *WHERE* clause.)
- Update pseudocolumns or expressions

There are some restrictions on how subviews and materialized views can be defined in Oracle:

- The subview must use aliases for **ROWID**, **ROWNUM**, or **LEVEL** pseudocolumns.
- The subview cannot query **CURVAL** or **NEXTVAL** pseudocolumns.
- The subview cannot contain the *SAMPLE* clause.
- The subview evaluates all columns of a *SELECT * FROM...* statement at compile time. Thus, any new columns added to the base table will not be retrieved by the subview until the view is recompiled.

Note that older versions of Oracle supported partitioned views. This feature has been deprecated. You should use explicitly declared partitions instead.

PostgreSQL

PostgreSQL supports a no-frills subset of the ANSI standard:

```
CREATE VIEW view_name [ (column [...]) ]
```

```
AS select_statement
```

PostgreSQL's *CREATE VIEW* does not support some of the more complex options that other vendors do. However, it does allow views to be built on tables and other defined class objects. PostgreSQL views are built only upon other tables, not upon other views. PostgreSQL views are always read-only and cannot be used to perform data modifications on the underlying base tables.

SQL Server

SQL Server supports some extensions to the ANSI standard, but does not offer object views or subviews:

```
CREATE VIEW view_name [(column [...])] 
```

```
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA} [...]]
```

```
AS
```

```
select_statement
```

```
[WITH CHECK OPTION]
```

SQL Server's implementation of *ALTER VIEW* allows you to change an existing view without affecting the permissions or dependent objects of the view:

```
ALTER VIEW view_name [(column [...])] 
```

```
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA} [...]]
```

```
AS
```

```
select_statement
```

```
[WITH CHECK OPTION]
```

The parameters are as follows:

ENCRYPTION

Encrypts the text of the view in the **syscomments** table. This option is usually invoked by software vendors who want to protect their intellectual capital.

SCHEMABINDING

Binds the view to a specific schema, meaning that all objects in the view must be referenced by their full name (both owner and object name). The view and any tables referenced in the view *SELECT* clause must be fully qualified, as in **pubs.scott.employee**. Views created with *SCHEMABINDING* (and tables referenced by these views) must have the schema binding dropped (via *ALTER VIEW*) before they may be dropped or altered.

VIEW_METADATA

Specifies that SQL Server return metadata about the view (rather than the base table) to calls made from DBLIB or OLEDB APIs. Views created or altered with *VIEW_METADATA* enable their columns to be updated by *INSERT* and *UPDATE INSTEAD OF* triggers.

WITH CHECK OPTION

Forces the view to accept only inserted and updated data that can be returned by the view's *SELECT* statement.

The *SELECT* clause of a SQL Server view cannot:

- Have the *COMPUTE*, *COMPUTE BY*, *INTO*, or *ORDER BY* clauses (*ORDER BY* is allowed if you use *SELECT TOP*)
- Reference a temporary table
- Reference a table variable
- Reference more than 1,024 columns, including those referenced by subqueries

Here, we define a SQL Server view with both *ENCRYPTION* and *CHECK OPTION*:

```
CREATE VIEW california_authors (last_name, first_name, author_id)
```

```
WITH ENCRYPTION
```

```
AS
```

```
    SELECT au_lname, au_fname, au_id
```

```
    FROM authors
```

```
    WHERE state = 'CA'
```

```
WITH CHECK OPTION
```

```
GO
```

SQL Server allows multiple *SELECT* statements in a view, as long as they are linked with *UNION* or *UNION ALL* clauses. SQL Server also allows functions and hints in the *SELECT* statement of a view. A SQL Server view is updateable if all of the items in the following list are true.

- The *SELECT* statement has no aggregate functions
- The *SELECT* statement does not contain *TOP*, *GROUP BY*, *DISTINCT*, or *UNION*
- The *SELECT* statement has no derived columns (see *SUBQUERY*)
- The *FROM* clause of the *SELECT* statement references at least one table

SQL Server allows indexes to be created on views (see *CREATE INDEX*). By creating a unique, clustered index on a view, you cause SQL Server to store a physical copy of the view on the database. Changes to the base table are automatically updated in the indexed view. Indexed views consume extra disk space, but provide a boost in performance. Indexed views must be built using the *SCHEMABINDING* clause.

SQL Server also allows the creation of local partitioned views and distributed partitioned views. A *local partitioned view* is a partitioned view where all views are present on the same SQL Server. A *distributed partitioned view* is a partitioned view where one or more views are located on remote servers.

Partitioned views must very clearly derive their data from different sources, with each distinct data source joined with a *UNION ALL* statement to the next. Furthermore, all columns of the partitioned views should be selected and identical.

(The idea is that you have split the data out logically by means of a frontend application. SQL Server then recombines the data through the partitioned view.) This example shows how the data in the view comes from three separate SQL Servers:

```
CREATE VIEW customers
AS
--Select from a local table on server 'New_York'
SELECT *
FROM sales_archive.dbo.customers_A
UNION ALL
SELECT *
FROM houston.sales_archive.dbo.customers_K
UNION ALL
SELECT *
FROM los_angeles.sales_archive.dbo.customers_S
```

Note that each remote server (*New_York*, *houston*, and *los_angeles*) has to be defined as a remote server on all of the SQL Servers using the distributed partitioned view. Partitioned views can greatly boost performance because they can split I/O and user loads across many machines. But they are also difficult to plan, create, and maintain. Be sure to read the vendor documentation for complete details about all the permutations available with partitioned views.

When altering an existing view, SQL Server acquires and holds an exclusive schema lock on the view until the alteration is finished. *ALTER VIEW* drops any indexes that might be associated with a view. You must manually recreate them using *CREATE INDEX*.

See Also

- CREATE/ALTER TABLE
- DROP
- SELECT
- SUBQUERY

DECLARE CURSOR Command

The *DECLARE* command is one of four commands used in cursor processing, along with *FETCH*, *OPEN*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than in a complete set. The *DECLARE CURSOR* command specifies the exact records to be retrieved and manipulated one row at a time from a specific table or view.

In other words, cursors are especially important for relational databases because databases are set-based, while most client-centric programming languages are row-based. This is important for two reasons. First, cursors allow programmers to program using methodologies supported by their favorite row-based programming language. Second, cursors run counter to the default behavior of some relational database platforms, which operate on a set of records and, on those specific platforms, may be noticeably slower than standard set-based operations.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

DECLARE *cursor_name* [{SENSITIVE | INSENSITIVE | ASENSITIVE}]

[[NO] SCROLL] CURSOR [{WITH | WITHOUT} HOLD]

[{WITH | WITHOUT} RETURN}]

FOR *select_statement*

[FOR {READ ONLY | UPDATE [OF *column* [,...]]}]

Keywords

DECLARE *cursor_name*

Gives the cursor a unique name in the context in which it is defined — for example, in the database or schema where it is created. The name must be unique, meaning that no other cursors may share the same name.

SENSITIVE | *INSENSITIVE* | *ASENSITIVE*

Defines the manner in which the cursor interacts with the source table and the result set retrieved by the cursor query, where:

SENSITIVE

Tells the database to operate directly against the result set, such that the cursor will see changes to its result set as it moves through the records.

INSENSITIVE

Tells the database to create a temporary but separate copy of the result set, so that all changes made against the result set by other operations are invisible to the cursor.

ASENSITIVE

Allows the database implementation to determine whether to make a copy of the result set. *ASENSITIVE* is the SQL2003 default.

[NO] SCROLL

Tells the database not to enforce processing one row at a time, using *FETCH NEXT*, but that all forms of the *FETCH* clause are allowed against the result set. *NO SCROLL* enforces processing one row at a time.

{WITH | WITHOUT} HOLD

Tells the cursor to remain open when a transaction encounters the *COMMIT* statement. Conversely, *WITHOUT HOLD* tells the cursor to close when the transaction encounters a *COMMIT* statement. (A cursor that uses *WITH HOLD* can only be and is always closed with a *ROLLBACK* statement or a *CLOSE CURSOR* statement.)

{WITH | WITHOUT} RETURN

Used only in a stored procedure, the *WITH RETURN* clause tells the database to return the result set, if it is still open, when the stored procedure terminates. With the *WITHOUT RETURN* clause, which is the default behavior, all open cursors are implicitly closed when the stored procedure terminates.

FOR *select_statement*

Defines the underlying *SELECT* statement that determines the rows in the result set of the cursor. As with a regular *SELECT* statement, the results of the query may be sorted according to an *ORDER BY* clause.

```
FOR {READ ONLY | UPDATE [OF column [,...]] }
```

Specifies that the cursor is not updateable in any way, using *FOR READ ONLY*. This is the default when the cursor is defined with *SCROLL* or *INSENSITIVE* properties or the query contains an *ORDER BY* clause or is against a non-updateable table. Alternately, you can specify *FOR UPDATE OF column1, column2 [,...]*, defining the columns where you want to execute *UPDATE* statements. You can leave off a column list on the *FOR UPDATE OF* to include all columns in the cursor.

Rules at a Glance

The *DECLARE CURSOR* command enables the retrieval and manipulation of records from a table one row at a time. This provides row-by-row processing, rather than the traditional set processing offered by SQL.

At the highest level, these steps are followed when working with a cursor:

- Create a cursor using *DECLARE*.
- Open the cursor using *OPEN*.
- Operate against the cursor using *FETCH*.
- Dismiss the cursor using *CLOSE*.

The *DECLARE CURSOR* command works by specifying a *SELECT* statement. Each row returned by the *SELECT* statement may be individually retrieved and manipulated. In this Oracle example, the cursor is declared in the declaration block along with some other variables. The cursor is then opened, fetched against, and then closed in the *BEGIN...END* block that follows:

DECLARE

```
CURSOR title_price_cursor IS
```

```
SELECT title, price
```

```
FROM titles
```

```
WHERE price IS NOT NULL;
```

```
title_price_val title_price_cursor%ROWTYPE;
```

```
new_price NUMBER(10,2);
```

```
BEGIN
```

```
OPEN title_price_cursor;
```

```
FETCH title_price_cursor INTO title_price_val;
```

```
new_price := "title_price_val.price" * 1.25
```

```
INSERT INTO new_title_price
```

```
VALUES (title_price_val.title, new_price)
```

```
CLOSE title_price_cursor;
```

```
END;
```

Because this example uses PL/SQL, much of the code is beyond the scope of this book. However, the *DECLARE* block clearly shows that the cursor is declared. In the PL/SQL execution block, the cursor is initialized with the *OPEN* command, values are retrieved with the *FETCH* command, and the cursor finally is terminated with the *CLOSE* command.

The *SELECT* statement is the heart of your cursor, so it is a good idea to test it thoroughly before embedding it in the *DECLARE CURSOR* statement. The *SELECT* statement may operate against a base table or a view. For that matter, read-only cursors may operate against non-updateable views. The *SELECT* statement can have subclauses like *ORDER BY*, *GROUP BY*, and *HAVING* if it is not updating the base table. If the cursor is *FOR UPDATE*, then it is a good idea to exclude these clauses from the *SELECT* statement.

Local cursors are sometimes used as output parameters of stored procedures. Thus, a stored procedure could define and populate a cursor and pass that cursor to a calling batch or stored procedure.

In this simple DB2 example, we'll declare a cursor that looks at department numbers, names, and manager numbers for all departments in the **admin_group 'X01'**:

```
EXEC SQL  
  
DECLARE dept_cursor CURSOR  
  
FOR SELECT dept_nbr, dept_name, mgr_nbr  
  
FROM department  
  
WHERE admin_group = 'X01'  
  
ORDER BY dept_name ASC, dept_nbr DESC, mgr_nbr DESC;
```

In this next example from Microsoft SQL Server, a cursor from the **publishers** table is declared and opened. The cursor takes the first record from **publishers** that matches the *SELECT* statement and inserts it into another table; it then moves to the next record and the next, until all records are processed. Finally, the cursor is closed and deallocated (*DEALLOCATE* is used only with Microsoft SQL Server):

```
DECLARE @publisher_name VARCHAR(20)  
  
DECLARE pub_cursor CURSOR  
  
FOR SELECT pub_name FROM publishers  
  
WHERE country <> 'USA'  
  
OPEN pub_cursor  
  
FETCH NEXT FROM pub_cursor INTO @publisher_name  
  
WHILE @@FETCH_STATUS = 0  
  
BEGIN  
  
INSERT INTO foreign_publishers VALUES(@publisher_name)  
  
END  
  
CLOSE pub_cursor  
  
DEALLOCATE pub_cursor
```

In this example, you see how the cursor moves through a set of records. (The example was intended only to demonstrate this concept, since there is actually a better way to accomplish the same task, namely an *INSERT...SELECT* statement.)

Programming Tips and Gotchas

Most platforms do not support dynamically executed cursors. Rather, cursors are embedded within an application program, stored procedure, user-defined function, etc. In other words, the various statements for creating and using a cursor are usually used only in the context of a stored procedure or other database-programming object, not in a straight SQL script.

For ease of programming and migration, don't use the *SCROLL* clause or the various sensitivity keywords. Some platforms support only forward-scrolling cursors, so you can avoid headaches later by just sticking to the simplest form of cursor.

Cursors behave differently on the various database platforms when crossing a transaction boundary — for example, when a set of transactions is rolled back in the midst of a cursor. Be sure to get familiar with exactly how each database platform behaves in these circumstances.

MySQL does not support server-side cursors in the ANSI SQL style, but does support extensive C-programming extensions that provide the same functionality.

DB2

DB2 supports the basic *DECLARE CURSOR* statement along with some variations on the *WITH HOLD* and *WITH RETURN* clauses, as in the following example:

```
DECLARE cursor_name CURSOR  
  
[WITH HOLD] [WITH RETURN [{TO CALLER | TO CLIENT}]]  
  
FOR select_statement
```

where:

WITH RETURN [{TO CALLER | TO CLIENT}]

Used only in a stored procedure, the *WITH RETURN* clause tells the database to return the result set, if it is still open, when the stored procedure terminates. (DB2 will accept *WITH RETURN* on cursors outside of a stored procedure, but it has no effect.) It has these options:

TO CALLER

Returns the result set to the caller, such as another stored procedure or process on the database server. It will also return the result set to a client, if that is what called the stored procedure. This is the default behavior for cursors embedded in DB2 external stored procedures.

TO CLIENT

Returns the result set to the client application, bypassing any intermediate processes such as nested stored procedures. This clause cannot be used to return result sets to functions or methods.

FOR *select_statement*

Defines the underlying *SELECT* statement that determines the rows in the result set of the cursor. The *SELECT* statement cannot include parameters, but can include host variables, as long as they have been declared before the cursor. You may also use a prepared *SELECT* statement as the *select_statement*, using the DB2 *PREPARE* command.

DB2 provides syntax to explicitly declare a cursor as read-only or updateable as part of the *SELECT* statement rather than as part of the *DECLARE CURSOR* statement.

When not specified, the updateability of a cursor is implicitly determined according to a variety of rules that improve locking. First of all, a cursor is assumed to be *read-only* if it is not deleteable. A DB2 cursor is *deleteable* when it has the *FOR UPDATE* clause plus:

- The *FROM* clause of the *SELECT* statement references only one base table or deleteable view.
- The *SELECT* statement does not contain a *VALUES* clause, a *DISTINCT* clause, a *GROUP BY* clause, a *HAVING* clause, an *ORDER BY* clause, a *FOR READ ONLY* clause, or *UNION*, *EXCEPT*, or *INTERSECT* clauses.
- The *SELECT* statement is free of column functions in the select item list.

The columns of a DB2 cursor are *updateable* when it meets the requirements to be deleteable plus:

- The columns are resolved to columns in the base table or view.
- The *LANGLEVEL* bind option is *MIA*, *SQL92E*, or the columns are referenced in the *FOR UPDATE* clause of the *select_statement*.



A DB2 cursor may also be updateable if it is statically defined or has the SQL-standard bind option (*MIA*) applied to *LANGLEVEL*. The *LANGLEVEL* setting *MIA* indicates that the cursor runs under SQL92 rules. The setting *SQL92E* indicates that the cursor runs under the SQL92 entry-level rules. And the setting *SAA1* indicates that the cursor runs under the DB2 common (old) rules. *LANGLEVEL* is assigned using the DB2 *PREP* command.

DB2 also allows cursors to be *ambiguous*. An ambiguous cursor is considered updateable unless the *BLOCKING* bind option is *ALL*, whence it is considered read-only. A cursor is ambiguous when all these conditions are true:

- The *select_statement* doesn't include a *FOR READ ONLY* or *FOR UPDATE* clause.
- The language level setting (*LANGLEVEL*) is set to the value *SAA1*.
- The *select_statement* is dynamically prepared.
- The cursor is otherwise deletable.

There are a couple things to be careful of with DB2 cursors:

- System function values, such as those returned by *CURRENT DATE*, *CURRENT TIME*, or *CURRENT TIMESTAMP*, are determined when a cursor is opened and not when the *FETCH* statement is executed. Therefore, any of these values would be identical for every record in the result set, since the values are determined once when the cursor is opened.
- If a *LANGUAGE* other than SQL (such as Java) is used for the stored procedure, the default behavior is *WITH RETURN TO CALLER*.

MySQL

Not supported.

Oracle

Oracle has a rather interesting implementation of cursors. In reality, all Oracle data modification statements (*INSERT*, *UPDATE*, *DELETE*, and *SELECT*) implicitly open a cursor. For example, a C program accessing Oracle would not need to issue a *DECLARE CURSOR* statement to retrieve data on a record-by-record basis because that is the implicit behavior of Oracle. Because of this behavior, *DECLARE CURSOR* is something you use only in PL/SQL constructs, like stored procedures, and not something you would use in a script that is purely SQL.



Since cursors can only be used in stored procedures and user-defined functions on Oracle, they are documented in the Oracle PL/SQL reference material, not in the SQL reference material.

Oracle utilizes a variation of the *DECLARE CURSOR* statement that supports parameterized inputs, as in the following syntax:

```
DECLARE CURSOR cursor_name [ ( parameter datatype [,...] ) ]
```

```
IS select_statement
```

```
[FOR UPDATE [OF column_name [,...]]]
```

where:

```
parameter datatype [,...] IS select_statement
```

Defines the *select_statement* used to retrieve the cursor result set. It serves the same purpose as the ANSI SQL2003 *FOR select_statement* clause.

```
FOR UPDATE [OF column_name]
```

Defines the cursor or specific columns of the cursor as updateable. Otherwise, the cursor is assumed to be read-only.

In Oracle, variables are not allowed in the *WHERE* clause of the *SELECT* statement unless they are first declared as variables. The parameters are not assigned at the *DECLARE*; instead, they are assigned values at the *OPEN* command. This is important since any system function will return an identical value for all rows in the cursor result set.

PostgreSQL

PostgreSQL does not support the *WITH* clauses. It does allow you to return result sets in *BINARY* rather than text format. Although the compiler will not cause errors with many of the ANSI keywords, PostgreSQL's implementation of *DECLARE CURSOR* is more limited than it might first appear:

```
DECLARE cursor_name [BINARY] [INSENSITIVE] [SCROLL] CURSOR
```

```
FOR select_statement
```

```
[FOR {READ ONLY | UPDATE [OF column_name [...]]}]
```

where:

BINARY

Forces the cursor to retrieve binary-formatted data rather than text-formatted data.

INSENSITIVE

Indicates that data retrieved by the cursor is unaffected by updates from other processes or cursors. However, this is PostgreSQL's default behavior, so adding or leaving this keyword off has no effect.

SCROLL

Allows multiple rows to be retrieved by a single *FETCH* operation. However, this is PostgreSQL's default behavior, so adding or leaving off this keyword has no effect.

```
FOR {READ-ONLY | UPDATE}
```

Indicates that a cursor is opened using read-only mode or updateable mode, respectively. However, PostgreSQL only supports read-only cursors. That means that the *FOR READ-ONLY* clause has no effect, while the *FOR UPDATE* (*column_name*) clause produces an informational error message.

PostgreSQL closes any existing cursor when a newly declared cursor is created with the same name. Binary cursors tend to be faster because PostgreSQL stores data as binary on the backend. However, user applications are only text aware. Therefore, you would have to build in binary handling for any *BINARY* cursors.

PostgreSQL allows cursors only within a transaction. You should enclose a cursor within a *BEGIN* and *COMMIT* or *ROLLBACK* transaction block.

PostgreSQL does not support an explicit *OPEN* cursor statement. Cursors are considered open when they are declared. So to declare and open a cursor in PostgreSQL, you could use code like this:

```
DECLARE pub_cursor CURSOR
```

```
FOR SELECT pub_name FROM publishers
```

```
WHERE country <> 'USA';
```

SQL Server

SQL Server supports the ANSI standard, as well as a good many extensions that provide flexibility in how a cursor scrolls through a result set and manipulates data, as in the following syntax:

```
DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR
```

[LOCAL | GLOBAL] [FORWARD_ONLY | SCROLL]

[STATIC | KEYSSET | DYNAMIC | FAST_FORWARD]

[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]

[TYPE_WARNING]

FOR *select_statement*

[FOR {READ ONLY | UPDATE [OF *column_name* [,...]]}]]

The parameters are as follows:

LOCAL | *GLOBAL*

Scopes the cursor for either the local Transact-SQL batch or makes the cursor available to all Transact-SQL batches issued by the current session via *OPEN* and *FETCH* statements, respectively. A global cursor name can be referenced by any stored procedure, function, or Transact-SQL batch executed in the current session. A global cursor is implicitly deallocated at disconnect, but a local cursor must be manually deallocated. The *LOCAL* and *GLOBAL* keywords are optional. The default behavior, if neither is specified, is defined by the *default to local* database property.

INSENSITIVE | *FORWARD_ONLY* | *SCROLL*

Determines how the cursor will move through the result set, with three options:

INSENSITIVE

Creates the result set as a table in **TEMPDB**. Changes to the base table are not reflected in the cursor result set. Not compatible with SQL Server extensions to the *DECLARE CURSOR* command like *LOCAL*, *GLOBAL*, *STATIC*, *KEYSET*, *DYNAMIC*, *FAST_FORWARD*, etc. Can only be used in a SQL92-style *DECLARE CURSOR* statement, such as *DECLARE sample CURSOR INSENSITIVE FOR select_statement FOR UPDATE*.

FORWARD_ONLY

Indicates that the cursor must scroll from the first to the last record in the result set and that *FETCH NEXT* is the only supported form of the *FETCH* statement. The cursor is assumed to be *DYNAMIC* unless *STATIC* or *KEYSET* are used. *FAST_FORWARD* is mutually exclusive of *FORWARD_ONLY*.

SCROLL

Enables all *FETCH* options (*ABSOLUTE*, *FIRST*, *LAST*, *NEXT*, *PRIOR*, and *RELATIVE*). Otherwise, only *FETCH NEXT* is available. If *DYNAMIC*, *STATIC*, or *KEYSET* are used, the cursor defaults to *SCROLL* behavior. *FAST_FORWARD* is mutually exclusive of *SCROLL*.

STATIC | *KEYSET* | *DYNAMIC* | *FAST_FORWARD*

Determines how records are manipulated in the result set. These settings are incompatible with the *FOR READ ONLY* and *FOR UPDATE* clauses. There are four options:

STATIC

Makes a temporary copy of the result set data and stores it in *tempdb*. Modifications made to the source table or view do not show up when processing the cursor. *STATIC* cursors cannot modify data in the source table or view.

KEYSET

Makes a temporary copy of the result set with membership and fixed row order (also known as a keyset) in *tempdb*. The keyset correlates the data in the cursor result set to the base table or view such

that the cursor can see changes made to the underlying data. Rows that have been deleted or updated show an @@FETCH_STATUS of -2 (unless the update was done using *UPDATE...WHERE CURRENT OF*, in which case they are fully visible), while rows inserted by other users are not visible at all.

DYNAMIC

Determines the records in the cursor result set as each *FETCH* operation is executed. Thus, *DYNAMIC* cursors see all changes made to the base table or view, even those by other users. Because the result set may be in constant flux, *DYNAMIC* cursors do not support *FETCH ABSOLUTE*.

FAST_FORWARD

Creates a *FORWARD_ONLY*, *READ_ONLY* cursor that quickly reads through the entire cursor result set at one time.

READ_ONLY | *SCROLL_LOCKS* | *OPTIMISTIC*

Determines concurrency and positional update behavior for the cursor. These settings are incompatible with *FOR READ ONLY* and *FOR UPDATE*. The allowable parameters include:

READ_ONLY

Prevents updates to the cursor and disallows the cursor from being referenced in *UPDATE* or *DELETE* statements that contain *WHERE CURRENT OF*.

SCROLL_LOCKS

Ensures that positional updates and deletes succeed by locking the cursor result set rows as they are read into the cursor. The locks are held until the cursor is closed and deallocated. *SCROLL_LOCKS* is mutually exclusive of *FAST_FORWARD*.

OPTIMISTIC

Ensures that positional updates and deletes succeed unless the row being updated or deleted in the cursor result set has changed since it was read into the cursor. SQL Server accomplishes this by comparing a timestamp, or a checksum value when no timestamp exists, on the columns. It does not lock the rows in the cursor result set. *OPTIMISTIC* is mutually exclusive of *FAST_FORWARD*.

TYPE_WARNING

Warns the user when the cursor is implicitly converted from one type to another — for example from *SCROLL* to *FORWARD_ONLY*.

FOR {READ ONLY |

Identifies the cursor as read only using the ANSI standard syntax. This keyword is not compatible with the other type identifiers shown previously. It should only be used with *INSENSITIVE*, *FORWARD_ONLY*, and *SCROLL*.

FOR UPDATE [OF *column_name* [,...]]}

Allows updates to columns in the cursor using *UPDATE* and *DELETE* statements with the *WHERE CURRENT OF* clause. If *FOR UPDATE* is used without a column list, then all columns in the cursor are updateable. Otherwise, only those columns listed are updateable.



Microsoft SQL Server allows two basic forms of syntax for the *DECLARE CURSOR* statement. *The syntax forms are not compatible!* The basic forms of syntax are SQL92 compatible and Transact-SQL extensions. You *cannot* mix keywords of the two forms together.

The SQL-92 compatibility syntax for *DECLARE CURSOR* is:


```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [,...] ] } ]
```

While the Transact-SQL extended syntax for *DECLARE CURSOR* is:

```
DECLARE cursor_name CURSOR
[ LOCAL | GLOBAL ] [ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [,...] ] ]
```

The SQL92 compatibility syntax is supported to enable your code to be more transportable. The Transact-SQL extensions syntax enables you to define cursors using the same cursor types that are available in popular database APIs like ODBC, OLE-DB, and ADO, but may only be used in SQL Server stored procedures, user-defined functions, triggers, and ad hoc queries.

If you do define a Transact-SQL extension cursor but do not define concurrency behavior using the *OPTIMISTIC*, *READ_ONLY*, or *SCROLL_LOCKS* keywords, then:

- The cursor defaults to *READ-ONLY* if the cursor is defined as *FAST_FORWARD*, *STATIC*, or the user has insufficient privileges to update the base table or view.
- The cursor defaults to *OPTIMISTIC* if the cursor has been defined as *DYNAMIC* or *KEYSET*.

Note that variables may be used in the *select_statement* of a SQL Server cursor, but the variables are evaluated when the cursor is declared. Thus, a cursor containing the column based on the system function *GETDATE*() will have the same date and time for every record in the cursor result set.

In the following SQL Server example, we use a *KEYSET* cursor to change blank spaces to dashes in the **phone** column of the **authors** table:

```
SET NOCOUNT ON

DECLARE author_name_cursor CURSOR LOCAL KEYSET TYPE_WARNING
    FOR SELECT au_fname FROM pubs.dbo.authors

DECLARE @name varchar(40)

OPEN author_name_cursor

FETCH NEXT FROM author_name_cursor INTO @name

WHILE (@@fetch_status <> -1)

BEGIN
-- @@fetch_status checks for 'record not found' conditions and errors

    IF (@@fetch_status <> -2)

        BEGIN

            PRINT 'updating record for ' + @name

            UPDATE pubs.dbo.authors

            SET phone = replace(phone, ' ', '-')

            WHERE CURRENT OF author_name_cursor
```

```
END  
  
FETCH NEXT FROM author_name_cursor INTO @name  
  
END  
  
CLOSE author_name_cursor  
  
DEALLOCATE author_name_cursor  
  
GO
```

See Also

CLOSE CURSOR
FETCH
OPEN

DELETE Statement

The *DELETE* statement erases records from a specified table or tables. *DELETE* statements acting against tables are sometimes called *search deletes*. The *DELETE* statement may also be used in conjunction with a cursor. *DELETE* statements acting upon the rows of a cursor are sometimes called *positional deletes*.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DELETE FROM { table_name | ONLY (table_name) }  
  
[ { WHERE search_condition | WHERE CURRENT OF cursor_name } ]
```

Keywords

FROM *table_name*

Identifies the table, called *table_name*, from which rows will be deleted. The *table_name* assumes the current schema if one is not specified. You may alternately specify a single table view name. *FROM* is mandatory, except in the *DELETE...WHERE CURRENT OF* statement. When not using the *ONLY* clause, do not enclose the *table_name* in parentheses.

ONLY (*table_name*)

Restricts cascading of the deleted records to any subtables of the target table or view. *ONLY* effects only typed (object-oriented) tables and views. If used with a non-typed table or view, it is ignored and does not cause an error. If *ONLY* is used, you must enclose the *table_name* in parentheses.

WHERE *search_condition*

Defines search criteria for the *DELETE* statement using one or more *search_condition* clauses to ensure that only the target rows are deleted. Any legal *WHERE* clause is acceptable. Typically, these criteria are evaluated against each row of the table before the deletion occurs.

WHERE CURRENT OF *cursor_name*

Restricts the *DELETE* statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

The *DELETE* statement erases rows from a table or view. Space released by erased rows will be returned to the database where the table is located, though it may not happen immediately.

A simple *DELETE* statement that erases all records in a given table has no *WHERE* clause, as in the following:

```
DELETE FROM sales;
```

You can use any valid *WHERE* clause to filter records that you do not want to delete. All three of the following examples show valid *DELETE* statements, and since all are search deletes, they all include the *FROM* clause:

```
DELETE FROM sales
```

```
WHERE qty IS NULL;
```

```
DELETE FROM suppliers
```

```
WHERE supplierid = 17
```

```
OR companyname = 'Tokyo Traders';
```

```
DELETE FROM distributors
```

```
WHERE postalcode IN
```

```
(SELECT territorydescription FROM territories);
```

In some cases, you may wish to delete a specific row that is being processed by a declared and open cursor:

```
DELETE titles WHERE CURRENT OF title_cursor;
```

The above query assumes that you have declared and opened a cursor named **title_cursor**. Whichever row the cursor is on will be deleted when the command, shown above, is executed. Note that in the positional delete shown earlier, the *FROM* clause is not required.

Programming Tips and Gotchas

It is rare to issue a *DELETE* statement without a *WHERE* clause, because this results in deleting *all* rows from the affected table. You should first issue a *SELECT* statement with the same *WHERE* clause you intend to use on the *DELETE* statement. That way you can be sure exactly which records will be deleted.

If it becomes necessary to remove all the rows in a table, you should consider using the non-ANSI *TRUNCATE TABLE* statement. In those databases that support the command, *TRUNCATE TABLE* is usually a faster method to physically remove all rows. *TRUNCATE TABLE* is faster than *DELETE* because the deletion of individual records is not logged. On some platforms, this makes rollback of a *TRUNCATE* statement impossible. The reduction of logging overhead saves considerable time when erasing a large number of records. On some database platforms, all foreign keys on the table must be dropped before issuing a *TRUNCATE* statement.

DB2

DB2 supports the ANSI standard, with the addition of transaction isolation control and the ability to assign an alias to the table from which records will be deleted, as in the following:

```
DELETE FROM { table_name | ONLY (table_name) } [AS alias]
```

[WHERE {*search_conditions* | CURRENT OF *cursor_name*}]

[WITH { RR | RS | CS | UR }]

The parameters are:

AS *alias*

Allows the target *table_name* to use an alias. This is especially useful for correlated subqueries. (Note that DB2 has a special object called an *alias* that may be created with the DB2 *CREATE ALIAS* statement. In this case, the term "alias" represents what the ANSI SQL standard calls a *correlation name*, something akin to a simple nickname for the table or view.)

WITH {RR | RS | CS | UR}

Controls transaction isolation level for the update statement. *RR* is repeatable read, *RS* is read stability, *CS* is cursor stability, and *UR* is uncommitted read. See the [SET TRANSACTION Statement](#) section for full details on these transaction isolation levels.

On DB2, the target may be a table, updateable view, or nickname. As the ANSI standard dictates, rows will be deleted from any subtables of the target table unless you use the *ONLY* keyword. If *ONLY* is used, you must enclose the *table_name* in parentheses.

If you use *ONLY* in combination with *WHERE CURRENT OF*, then the *SELECT* statement that defined the cursor must use the *FROM ONLY* clause as well. For example:

```
DECLARE CURSOR emp_cursor FOR
```

```
  SELECT name, address
```

```
  FROM ONLY (employee);
```

```
DELETE FROM ONLY (employee)
```

```
  WHERE CURRENT OF emp_cursor;
```

The previous code assumes that **employee** is the base table, with subtables below it. The cursor will only select rows that are contained in **employee** and not the other types contained in the subtables (e.g., **parttime**, **salaried**, **student**). Similarly, deletes will only take place at this level of the hierarchy as well.

MySQL

MySQL allows a number of extensions over the ANSI standard, but does not support the *WHERE CURRENT OF* clause, as in the following:

```
DELETE [LOW_PRIORITY] [QUICK] [table_name[*] [...] ]
```

```
{FROM table_name[*] [...] | [USING table_name[*] [...] ] }
```

```
[WHERE search_condition]
```

```
[ORDER BY clause]
```

```
[LIMIT nbr_of_rows]
```

The parameters are:

LOW PRIORITY

Delays the execution of *DELETE* until no other clients are reading from the table.

QUICK

Prevents the storage engine from merging index leaves during the delete operation.

DELETE *table_name* [,...]

Enables you to delete from more than one table at a time. Tables listed before the *FROM* clause, assuming one or more tables appear in the *FROM* clause, will be the target of the delete operation. If more than one table appears before the *FROM* clause, then all matching records in all tables will be deleted.

FROM *table_name*[.*]

Specifies the table or tables where records will be deleted. (The .* clause is an option to improve compatibility with MS-Access.) If tables are listed before the *FROM* clause, then the table or tables in the *FROM* clause are assumed to be used to support a join or lookup operation.

USING *table_name*[.*] [,...]

Substitute the table or tables before the *FROM* clause with those after the *FROM* clause.

ORDER BY *clause*

Specifies the order that rows will be deleted in. This is useful only in conjunction with *LIMIT*.

LIMIT *nbr_of_row*

MySQL also can place an arbitrary cap on the number of records deleted before control is passed back to the client using the *LIMIT nbr_of_rows* clause.

MySQL allows deletion from more than one table at a time. For example, the following two *DELETE* statements are functionally equivalent:

```
DELETE orders FROM customers, orders
```

```
WHERE customers.customerid = orders.customerid
```

```
AND orders.orderdate BETWEEN '19950101' AND '19951231'
```

```
DELETE FROM orders USING customers, orders
```

```
WHERE customers.customerid = orders.customerid
```

```
AND orders.orderdate BETWEEN '19950101' AND '19951231'
```

In the examples above, we delete all *orders* made by *customers* during the year 1995. Note that you cannot use *ORDER BY* or *LIMIT* clauses in a multitable delete like those shown above.

You can also delete records in orderly batches using the *ORDER BY* clause in conjunction with the *LIMIT* clause:

```
DELETE FROM sales
```

```
WHERE customerid = 'TORTU'
```

```
ORDER BY customerid
```

```
LIMIT 5
```

MySQL exhibits several behaviors that speed up the delete operation. For example, MySQL returns the number of records deleted when it completes the operation. However, MySQL will return zero as the number of deleted records when you delete all records from a table because it is faster for MySQL to return zero than to count the rows deleted. In *AUTOCOMMIT* mode, MySQL will even substitute a *TRUNCATE* statement for a *DELETE* statement without a *WHERE* clause because *TRUNCATE* is faster.

Note that the speed of a MySQL delete operation is directly related to the number of indexes on the table and the available index cache. You can speed up delete operations by executing the command against tables with few or no indexes or by increasing the size of the index cache.

Oracle

Oracle allows you to delete rows from tables, views, materialized views, nested subqueries, and partitioned views and tables, as follows:

DELETE [FROM]

```
{table_name | ONLY (table_name)} [alias]  
[({PARTITION (partition_name) |  
SUBPARTITION (subpartition_name)} ) |  
(subquery [WITH {READ ONLY |  
CHECK OPTION [CONSTRAINT constraint_name]} ] ) |  
TABLE (collection_expression) [ (+) ] }  
[WHERE search_condition]  
[RETURNING expression [,...] INTO variable [,...]
```

The parameters are:

table_name [*alias*]

Specifies the table, view, materialized view, or partitioned table or view where records will be deleted. You may optionally prepend a schema identifier to the *table_name* or append a database link. Otherwise, Oracle will assume the user's default schema and the local database server. You may optionally apply an *alias* to the *table_name*. The alias is required if the target table references an object type attribute or method.

PARTITION *partition_name*

Applies the delete operation to the named partition, rather than to the entire table. You are not required to name a partition when deleting from a partitioned table, but it can, in many cases, help reduce the complexity of the *WHERE* clause.

SUBPARTITION *subpartition_name*

Applies to a named subpartition, rather than to the entire table.

(*subquery* [WITH {READ ONLY | CHECK OPTION [CONSTRAINT *constraint_name*]}])

Specifies that the target for deletion is a nested subquery, not a table, view, or other database object. The parameters of this clause are:

subquery

Describes the *SELECT* statement that makes up the subquery. The subquery can be any standard subquery, though it may not contain an *ORDER BY* clause.

WITH READ ONLY

Specifies that the *subquery* cannot be updated.

WITH CHECK OPTION

Tells Oracle to abort any changes to the deleted table that would not appear in the result set of the subquery.

CONSTRAINT *constraint_name*

Tells Oracle to further restrict changes based upon a specific constraint identified by *constraint_name*.

TABLE (*collection_expression*) [(+)]

Informs Oracle that the *collection_expression* should be treated like a table though it may, in fact, be a subquery, a function, or some other collection constructor. In any case, the value returned by the *collection_expression* must be a nested table or *VARRAY*.

RETURNING *expression*

Retrieves the rows affected by the command, where *DELETE* normally only shows the number of rows deleted. The *RETURNING* clause can be used when the target is a table, a materialized view, or a view with a single base table. When used for single-row deletes, the *RETURNING* clause stores values from the row deleted by the statement, defined by *expression*, into PL/SQL variables and bind variables. When used for a multirow delete, the *RETURNING* clause stores the values of the deleted rows, defined by *expression*, into bind arrays.

INTO *variable*

Specifies the variables into which the values returned as a result of the *RETURNING* clause are stored. There must be a corresponding *variable* for every *expression* in the *RETURNING* clause.

When you execute a *DELETE* statement, Oracle releases space from the target table (or the base table of a target view) back to the table or index that owned the data.

When deleting from a view, the view cannot contain a set operator, the *DISTINCT* keyword, joins, an aggregate function, an analytic function, a subquery in the *SELECT* list, a collection expression in the *SELECT* list, a *GROUP BY* clause, an *ORDER BY* clause, a *CONNECT BY* clause, or a *START WITH* clause.

Here's an example where we delete records from a remote server:

```
DELETE FROM scott.sales@chicago;
```

In the following query, we delete from a derived table that is a collection expression:

```
DELETE TABLE(SELECT contactname FROM customers
               c WHERE c.customerid = 'BOTTM') s
WHERE s.region IS NULL OR s.country = 'MEXICO';
```

Here's an example where we delete from a partition:

```
DELETE FROM sales PARTITION (sales_q3_1997)
WHERE qty > 10000;
```

Finally, in the example below, we use the *RETURNING* clause to look at the values that are deleted:

```
DELETE FROM employee
WHERE job_id = 13
      AND hire_date + TO_YMINTERVAL('01-06') =< SYSDATE;
RETURNING job_lvl
INTO :int01;
```

The previous example deletes records from the **employee** table and returns the **job_lvl** values into the predefined **:into1** variable.

PostgreSQL

PostgreSQL uses the *DELETE* command to remove rows and any defined subclasses from the table. It is otherwise identical to the ANSI standard, as follows:

```
DELETE [FROM] [ONLY] table_name  
[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

When deleting rows from only the table specified, use the optional *ONLY* clause. Otherwise, PostgreSQL will also delete records from any explicitly defined subtable.

To delete all records from the **titles** table:

```
DELETE titles
```

To delete all records in the **authors** table where the last name starts with "Mc":

```
DELETE FROM authors  
WHERE au_lname LIKE 'Mc%'
```

To delete all **titles** with an old ID number:

```
DELETE titles WHERE title_id >= 40
```

To delete all **titles** that have no sales:

```
DELETE titles WHERE ytd_sales IS NULL
```

To delete all records in one table based on the results of a subquery against another table (in this case, the records that have a match concerning "computers" in the **titles** table are erased from the **titleauthor** table):

```
DELETE FROM titleauthor  
WHERE title_id IN  
(SELECT title_id  
FROM titles  
WHERE title LIKE '%computers%')  
DISCONNECT
```

SQL Server

Microsoft SQL Server allows records to be deleted both from tables and from views that describe a single table. SQL Server also allows a second *FROM* clause to allow *JOIN* constructs, as in the following example:

```
DELETE [TOP number [PERCENT]] [FROM] table_name [[AS] alias]  
[FROM table_source [...]]  
[ [{INNER | CROSS | [{LEFT | RIGHT | FULL} OUTER}]  
JOIN joined_table ON condition][,...]  
[WHERE search_condition | WHERE CURRENT OF [GLOBAL] cursor_name ]  
[OPTION (query_hint[,...n])]
```

The syntax elements are as follows:

DELETE *table_name*

Allows the deletion of records from the named table or view filtered by the *WHERE* clause. You can delete records from a view providing the view is based on one table and contains no aggregate functions and no derived columns. If you leave off the server name, database name, or schema name when naming the table or view, SQL Server assumes the current context. An *OPENDATASOURCE* or *OPENQUERY* function, as described in the [SELECT Statement](#) section may be referenced instead of a table or view.

TOP *number* [PERCENT]

Specifies that only the *number* of rows are deleted by the statement. If *PERCENT* is specified, only the first *number* percent of the rows are retrieved. If the *number* is an expression, such as a variable, it must be enclosed in parentheses. The expression should be of *FLOAT* datatype with a range of 0 to 100 when using *PERCENT*. When not using *PERCENT*, the expression should be of *BIGINT* datatype.

FROM *table_source*

Names an additional *FROM* clause that correlates records from the table in the first *FROM* clause using a *JOIN* rather than forcing you to use a correlated subquery. One or more tables may be listed in the second *FROM* clause.

[{INNER | CROSS | [{LEFT | RIGHT | FULL} OUTER]}] JOIN *joined_table* ON *condition*[,...]

Specifies one or more *JOIN* clauses, in conjunction with the second *FROM* clause. You may use any of the join types that SQL Server supports. Refer to the [JOIN Subclause](#) and [SELECT Statement](#) sections for more information.

GLOBAL *cursor_name*

Specifies that the delete operation should occur on the current row of an open global cursor. This clause is otherwise the same as the standard for *WHERE CURRENT OF*.

OPTION (*hint* [...])

Replaces elements of the default query plan with your own. Because the optimizer usually picks the best query plan for any query, we strongly discourage you from placing optimizer hints into your queries.

The primary addition to *DELETE* that SQL Server offers over and above the ANSI standard is a second *FROM* clause. The second *FROM* allows the use of the *JOIN* statement and makes it quite easy to delete rows from the table in the first *FROM* by correlating rows of a table declared in the second *FROM*. For example, you can use a rather complex subquery to erase all the **sales** records of computer books with this command:

```
DELETE sales
WHERE title_id IN
    (SELECT title_id
     FROM titles
     WHERE type = 'computer')
```

But SQL Server allows a more elegant construction using a second *FROM* clause and a *JOIN* clause:

```
DELETE s
FROM sales AS s
INNER JOIN titles AS t ON s.title_id = t.title_id
    AND type = 'computer'
```

The following example deletes all rows with an **order_date** of 2003 or earlier in batches of 2,500:

```
WHILE 1 = 1  
  
BEGIN  
  
    DELETE TOP (2500)  
  
    FROM sales_history WHERE order_date <= '20030101'  
  
    IF @@rowcount < 2500 BREAK  
  
END
```

The *TOP* clause should be used in favor of the old *SET ROWCOUNT* statement because the *TOP* clause is open to many more query optimization algorithms. (Prior to SQL Server 2005, the *SET ROWCOUNT* statement was often used with data modification transactions to process large numbers of rows in batches, thus preventing the transaction log from filling up and ensuring that row locks would not escalate to a full table lock.)

See Also

INSERT
SELECT
TRUNCATE TABLE
UPDATE

DISCONNECT Statement

The *DISCONNECT* statement terminates one or more connections created between the current SQL process and the database server.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DISCONNECT {CURRENT | ALL | connection_name | DEFAULT}
```

Keywords

CURRENT

Closes the currently active user connection.

ALL

Closes all open connections for the current user.

Rules at a Glance

DISCONNECT is used to disconnect a named SQL session (*connection_name*), the *CURRENT* connection, the *DEFAULT*

connection, or *ALL* connections held by the user. For example, we can disconnect a single session called `new_york`:

```
DISCONNECT new_york
```

In the next example, we might want to disconnect all currently open sessions for the current user process:

```
DISCONNECT ALL
```

Programming Tips and Gotchas

DISCONNECT is not universally supported across platforms. Do not build cross-platform applications based on *DISCONNECT* unless you've made provisions to disconnect SQL sessions using each platform's preferred disconnection methodology.

DB2

DB2 supports the ANSI standard for *DISCONNECT* in embedded SQL only. *DISCONNECT* can drop one or more connections only after a commit or rollback operation succeeds for the sessions to be dropped. You can also use the *RELEASE* statement to place a connection in a disconnect-pending condition.

MySQL

Not supported.

Oracle

Oracle allows *DISCONNECT* only in its ad hoc query tool, SQL*Plus, using this syntax:

```
DISC[ONNECT]
```

In this usage, the command ends the current session with the database server but otherwise allows work in SQL*Plus to continue. For example, a programmer can continue to edit the buffer, save run files, and so on. However, you must establish a new connection to issue any SQL commands. Exiting SQL*Plus and returning to the filesystem requires the *EXIT* or *QUIT* commands. For example, to end the current connection with an Oracle server:

```
DISCONNECT;
```

PostgreSQL

Not supported. Instead, every programming interface supports a disconnect operation; for example, *SPI_FINISH* is available under the Server Programming Interface, and *PG_CONNECT* is available under the PL/TCL programming package.

SQL Server

Microsoft SQL Server supports the ANSI syntax for *DISCONNECT* in Embedded-SQL (ESQL) only, not within its ad hoc querying tool, SQL Query Analyzer. To disconnect cleanly from Microsoft SQL Server in an ESQL program, use the *DISCONNECT ALL* statement.

See Also

CONNECT

DROP Statements

All of the database objects created with *CREATE* statements may be destroyed using complementary *DROP* statements. On some platforms, a *ROLLBACK* statement after a *DROP* statement will recover the dropped object. However, on other

database platforms, the *DROP* statement is irreversible and permanent, so it is advisable to use the command with care.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

Currently, the SQL2003 standard supports the ability to drop a lot of object types that are largely unsupported by most vendors. The ANSI SQL2003 syntax follows this format:

DROP {object_type} object_name {RESTRICT | CASCADE}

Keywords

DROP object_type object_name

Irreversibly and permanently destroys the specified object, called *object_name*, of the type identified. The *object_name* does not need a schema identifier, but the current schema is then assumed. ANSI SQL2003 supports a long list of object types, each created with their own corresponding *CREATE* statement. *CREATE* statements covered in this book with corresponding *DROP* statements include:

DOMAIN	SCHEMA
FUNCTION	TABLE
METHOD	TRIGGER
PROCEDURE	TYPE
ROLE	VIEW

RESTRICT | CASCADE

Prevents the *DROP* from taking place if any dependent objects exist (*RESTRICT*), or causes all dependent objects to also be dropped (*CASCADE*). This clause is not allowed with some forms of *DROP*, like *DROP TRIGGER*, but is mandatory for others, like *DROP SCHEMA*. To further explain, *DROP SCHEMA RESTRICT* will only drop an empty schema. Otherwise, if the schema contains objects, the operation will be prevented. In contrast, *DROP SCHEMA CASCADE* will drop a schema and all objects contained therein.

Rules at a Glance

For rules about the creation or modification of each of the object types, refer to the corresponding *CREATE/ALTER* statements.

The *DROP* statement destroys a pre-existing object. The object is permanently destroyed, and all users who had access to the object immediately lose the ability to access the object.

The object may be qualified — that is, you may fully specify the schema where the dropped object is located. For example:

DROP TABLE scott.sales_2004 CASCADE;

The above statement would not only drop the table **scott.sales_2004**, but also any views, triggers, or constraints built on it. On the other hand, a *DROP* statement may include an unqualified object name, and then the current schema context is assumed. For example:

```
DROP TRIGGER before_ins_emp;
```

```
DROP ROLE sales_mgr;
```

Although not required by the SQL2003 standard, most implementations cause the *DROP* command to fail if the database object is in use by another user.

Programming Tips and Gotchas

DROP will only work when it is issued against a pre-existing object of the appropriate type where the user has appropriate permissions, usually the *DROP TABLE* permission. (Refer to the [GRANT Statement](#) section for more information.) The SQL standard requires only that the owner of an object may drop it, but most database platforms allow variations on that requirement. For example, the database superuser can usually drop any object on the database server.

With some vendors, the *DROP* command fails if the database object has extended properties. For example, Microsoft SQL Server will not drop a table that is replicated unless you first remove the table from replication.



It is important to be aware that most vendors do not notify you if the *DROP* command creates a dependency problem. Thus, if a table that is used by a few views and by stored procedures elsewhere in the database is dropped, no warning is issued. Those other objects simply return failures when they are accessed. To prevent this problem, you may wish to use the *RESTRICT* syntax where it is available.

You may have noticed that the ANSI standard does not support certain common *DROP* commands like *DROP DATABASE* and *DROP INDEX*, even though every vendor covered in this book (and just about every one in the market) supports these commands. The exact syntax for each of these commands is covered in the platform-specific sections below.

DB2

DB2 does not support the *CASCADE* option. It supports many of the ANSI object types in addition to many of its own object types, as follows:

```
DROP {BUFFERPOOL | EVENT MONITOR | [SPECIFIC] FUNCTION |  
      FUNCTION MAPPING | INDEX | INDEX EXTENSION |  
      [SPECIFIC] METHOD | NICKNAME | DATABASE PARTITION GROUP |  
      PACKAGE | [SPECIFIC] PROCEDURE | SCHEMA | SEQUENCE | SERVER |  
      TABLE [HIERARCHY] | TABLESPACE[S] | TRANSFORM[S]  
      [{ALL | group_name}] FOR | TRIGGER | [DISTINCT] TYPE |  
      TYPE MAPPING | USER MAPPING FOR {authorization_name | USER} SERVER |  
      VIEW [HIERARCHY] | WRAPPER } object_name [RESTRICT]
```

The rules for DB2's *DROP* statements are less consistent than the ANSI standard's rules, so the full syntax of each *DROP* variant is shown below:

```
BUFFERPOOL bufferpool_name
```

Drops the named bufferpool as long as no tablespaces are assigned to it. Memory is recovered immediately, but disk space may be recovered at the next connection to the database.

EVENT MONITOR *event_monitor_name*

Drops the named event monitor as long as the event monitor is not set ON.

[SPECIFIC] FUNCTION *function_name* [(*datatype* [,...])] [RESTRICT]

Drops the named user-defined function. DB2 allows multiple functions of the same name. To drop these, use the syntax *DROP FUNCTION function_name (data type [,...])*, providing *data type* identifiers sufficient to uniquely identify the user-defined function. Use the optional *SPECIFIC* keyword to identify functions by the unique name assigned to each at creation time. You may drop a function while it is in use without causing the user to experience a failure (until they try to access it later). However, you *cannot* drop a function that is a component of a package, triggers, views, etc. Here are examples of all three syntaxes:

DROP FUNCTION sales_summary_1992

DROP FUNCTION sales_summary (int, float)

DROP SPECIFIC FUNCTION products.inventory_level

FUNCTION MAPPING *function_mapping_name*

Drops the named function mapping.

INDEX [EXTENSION] *index_name* [RESTRICT]

Drops the named index or, optionally, an index extension, as long as it is not a primary key or unique constraint. You should use *ALTER TABLE* to drop a primary key or unique constraint. *RESTRICT* is mandatory when dropping an index extension.

[SPECIFIC] METHOD *method_name* [(*datatype* [,...])]

[FOR *type_name*][RESTRICT]

Drops the named method body or specific method body, except those implicitly created by *CREATE TYPE*. The method specification remains as part of the definition of the subject type, unless removed with *ALTER TYPE DROP METHOD*. You may alternately use the syntax *DROP METHOD method_body_name (data type1 [,...])* to further identify the method via datatypes. The *FOR type_name* clause must be specified when *SPECIFIC* is not specified in order to name the type where the method will be dropped. All dependencies on a method must be dropped before the method can be dropped.

NICKNAME *nickname*

Drops the named nickname. Certain other dependent objects are also dropped, including materialized query tables, index specifications, views (not dropped but marked inoperable), and packages (not dropped but marked inoperable).

DATABASE PARTITION GROUP *db_prtn_name*

Drops the named database partition group, and all tablespaces and subordinant objects defined under it (such as tables, packages, constraints, etc.) are dropped or marked inoperable. You should drop any tablespaces from the database partition group first. For example:

DROP TABLESPACE sales_2004

DROP TABLESPACE sales_2003

DROP DATABASE PARTITION GROUP sales_archive_0304

PACKAGE *package_name* [VERSION *version_id*]

Drops the named package of *schema_name.package_id*. You may also optionally add the clause *VERSION version_id* after the *package_id* to further identify the package.

[SPECIFIC] PROCEDURE *procedure_name* [(*datatype1* [,...])] [RESTRICT]

Drops the named procedure or specific procedure. DB2 allows multiple procedures with the same name, each distinguished by their different datatype usage. You may alternately identify the procedure to drop via the syntax *DROP PROCEDURE procedure_name (datatype [,...])* syntax.

SCHEMA *schema_name* RESTRICT

Drops the named schema. The *RESTRICT* keyword requires that the schema be empty before it can be dropped.

SEQUENCE *sequence_name* [RESTRICT]

Drops the named sequence from the server. However, you cannot drop a sequence created for an *IDENTITY* column using the *DROP SEQUENCE* statement.

SERVER *server_name*

Drops the definition of a named data source from the catalog as well as all nicknames for objects residing on the remote server. In addition, index specifications, user-defined functions, user-defined type mappings, and user mappings dependent on the remote server are dropped. Packages and function mappings dependent on the remote server are invalidated.

TABLE [HIERARCHY] *table_name*

Drops the named base table, declared temporary table, materialized query table, or staging table from the server. Indexes, constraints, materialized query tables, and staging tables that reference a dropped table are also dropped. Subtables of a typed table are not automatically dropped and must be dropped before the parent table can be dropped. Views, packages, and triggers that reference the dropped table are rendered inoperable. Any DATALINK columns are unlinked, though asynchronously. The *HIERARCHY* option is used to drop a named type table hierarchy (i.e., both the parent table and all of its subtables). In this syntax, the **object_name** is the name of the root table in the typed table hierarchy. A table cannot be dropped if it has the *RESTRICT ON DROP* attribute.

TABLESPACE[S] *tablespace_name* [,...]

Drops one or more named tablespaces and all objects from those tablespaces, unless a table stored in the tablespace spans the tablespace to be dropped and at least one of the tablespaces spanned, or the table has the *RESTRICT ON DROP* property. The *TABLESPACE* keyword and the *TABLESPACES* keyword are synonyms.

TRANSFORM[S] {ALL | *group_name*} FOR *type_name*

Drops all transform groups for a user-defined datatype using the syntax *DROP TRANSFORM ALL FOR datatype_name*. Note that only the transform group and not the user-defined datatype are dropped. Alternately, use the syntax *DROP TRANSFORM transform_group_name FOR datatype_name* to drop a single transform group from a user-defined datatype. For example:

DROP TRANSFORM sales_calcs FOR orders

TRIGGER *trigger_name*

Drops the named trigger. This may also cause certain packages to be marked invalid.

[DISTINCT] TYPE *type_name* [RESTRICT]

Drops the named user-defined type or, with the optional *DISTINCT* keyword, drops the named distinct type. Using the optional *RESTRICT* keyword prevents the type from being dropped, if the type has any dependency such as being used in a table definition or in a view, has subtypes, etc.

TYPE MAPPING *type_mapping_name*

Drops the named user-defined type mapping, though no other subordinant objects are dropped.

USER MAPPING FOR {*user_name* | USER} SERVER *server_name*

Drops the user mapping to a federated server for a named user. If no server is specified, all user mappings are dropped for the named user. For example, the user **Dylan** no longer needs the **oracle50** data source:

```
DROP USER MAPPING FOR dylan SERVER oracle50
```

VIEW [HIERARCHY] *view_name*

Drops the named view from the schema. If the view is a parent object, then all subviews must be dropped before the parent view. Other objects that directly or indirectly depend on the view, such as triggers or materialized query tables, are marked inoperable. The optional *HIERARCHY* keyword tells DB2 to drop a parent typed view and all of its subviews.

WRAPPER *wrapper_name*

Drops the named wrapper as well as any server definitions, user-defined function mappings, and user-defined datatype mappings. This can cause a cascading action when dropping a server definition because objects dependent on the server definition (things like user mappings and nicknames) are also dropped. If the server definition results in a nickname being dropped, then objects dependent on the nickname (like views or packages) become inoperable.

RESTRICT

Optionally allowed on *FUNCTION*, *SPECIFIC FUNCTION*, *METHOD*, *SPECIFIC METHOD*, *PROCEDURE*, *SPECIFIC PROCEDURE*, *SEQUENCE*, *TYPE*, and *DISTINCT TYPE*. Mandatory for *INDEX EXTENSION* and *SCHEMA*. Like the ANSI standard, *RESTRICT* prevents the object from being dropped if a dependant object exists. The *RESTRICT* behavior is the default.

If you do not specify a schema along with the object name when performing a drop, DB2 will assume the current schema in dynamic SQL statements. DB2 will apply the *QUALIFIER* precompile/bind option for static SQL statements.

MySQL

MySQL supports a very limited number of objects via the *CREATE/ALTER* statement. Consequently, the syntax for *DROP* is also limited, as in the following:

```
DROP { DATABASE | [TEMPORARY] TABLE | INDEX } object_name [,...]
```

```
    [ON table_name]
```

```
    [IF EXISTS] object_name [,...] [RESTRICT [CASCADE]]
```

The syntax elements are:

DATABASE *database_name*

Drops the named database, including all the objects it contains, such as tables and indexes. The *DROP DATABASE* command removes all database and table files from the filesystem, as well as two-digit subdirectories. MySQL will return a message showing how many files were erased from the database directory. (Files of these extensions are erased: *.BAK*, *.DAT*, *.frm*, *.HSH*, *.ISD*, *.ISM*, *.MRG*, *.MYD*, *.MYI*, *.dm*, and *.fm*.) If the database is linked, both the link and the database are erased. You may only drop one database at a time. *RESTRICT* and *CASCADE* are not valid on *DROP DATABASE*.

[TEMPORARY] TABLE *table_name* [,...]

Drops one or more named tables, with table names separated from each other by commas. MySQL erases each table's definition and deletes the three table files from the filesystem (*.frm*, *.MYD*, and *.MYI*). Issuing this command causes MySQL to commit all active transactions. The *TEMPORARY* keyword drops only temporary tables without committing running transactions or checking access rights.

INDEX *index_name* ON *table_name*

Drops the named index from the named table on Version 3.22 or later. This command doesn't do anything on versions earlier than 3.22, but on 3.22 it actually executes an `ALTER TABLE...DROP INDEX` command behind the scenes. You cannot use `RESTRICT`, `CASCADE`, or `IF EXISTS` with a `DROP INDEX` statement.

`RESTRICT` | `CASCADE`

Acts as noise. These keywords do not generate an error by MySQL, nor do they do anything.

`IF EXISTS`

Prevents an error message when attempting to drop an object that does not exist. Usable in MySQL Version 3.22 or later.

MySQL supports *only* the ability to drop a database, a table (or tables), or an index from a table. Although the `DROP` statement will not fail with the `RESTRICT` and `CASCADE` optional keywords, they also don't do anything. You can use the `IF EXISTS` clause to prevent MySQL from returning an error message by trying to delete an object that doesn't exist.

Oracle

Oracle supports most of the ANSI keywords for the `DROP` statements, as well as many additional keywords corresponding to objects uniquely supported by Oracle, as follows:

`DROP {CLUSTER | CONTEXT | DATABASE | DATABASE LINK | DIMENSION |`

`DIRECTORY | FUNCTION | INDEX | INDEXTYPE | JAVA |`

`LIBRARY | MATERIALIZED VIEW | MATERIALIZED VIEW LOG |`

`OPERATOR | OUTLINE | PACKAGE | PROCEDURE | PROFILE |`

`ROLE | ROLLBACK SEGMENT | SEQUENCE | SYNONYM | TABLE |`

`TABLESPACE | TRIGGER | TYPE | TYPE BODY | USER | VIEW }`

object_name

The rules for Oracle `DROP` statements are less consistent than the ANSI standard's rules, so the full syntax of each `DROP` variant is shown below:

CLUSTER *cluster_name* [INCLUDING TABLES [CASCADE CONSTRAINTS]]

Drops the named cluster from the database. (Note that this drops a cluster; it does not uncluster a specific table from a cluster.) `INCLUDING TABLES` drops all tables that belong to the cluster. `CASCADE CONSTRAINTS` drops all referential integrity constraints that refer to the tables in the cluster. `DROP CLUSTER` will fail without `CASCADE CONSTRAINTS` when tables outside of the cluster have constraints (like a primary or foreign key) that refer to tables in the cluster. For example:

`DROP CLUSTER hr INCLUDE TABLES CASCADE CONSTRAINTS;`

CONTEXT *namespace*

Drops the named context from the database. Users currently in the namespace are unaffected, but will be unable to access the context when they next attempt to use that context. For example:

`DROP CONTEXT hr_context`

`DATABASE`

Drops a database from the Oracle server.

[PUBLIC] DATABASE LINK *database_link_name*

Drops the named database link from the database. You must include the *PUBLIC* keyword to drop a public database link. Note that the database link cannot be schema qualified. For example, to drop the database link called *hq.southeast.tenn*:

```
DROP DATABASE LINK hq.southeast.tenn;
```

DIMENSION *dimension_name*

Drops the named dimension. This does not invalidate materialized views built on the dimension, though they may execute more slowly and query requests may be rewritten.

DIRECTORY *directory_name*

Drops the named directory object from the database. This should not be used on directories containing files accessed by PL/SQL or OCI programs. For example, you could drop the *image_files* directory:

```
DROP DIRECTORY image_files;
```

FUNCTION *function_name*

Drops the named function, as long as it is not a component of a package. If you want to drop a function from a package, use the *CREATE PACKAGE...OR REPLACE* statement to redefine the package without the function you want to be rid of. Any local objects that depend on or call the function are invalidated and any statistical types associated with the function are disassociated.

INDEX *index_name* [FORCE]

Drops a named index or domain index from the database. Dropping an index invalidates all objects that depend on the parent table, including views, packages, functions, and stored procedures. Dropping an index invalidates cursors and execution plans that use the index, and will force a hard parse of the affected SQL statements when they are next executed.

Non-IOT indexes are secondary objects and can be dropped and recreated without any loss of user data. IOTs, because they combine both table and index data in the same structure, cannot be dropped and recreated in this manner. IOTs should be dropped using the *DROP TABLE* syntax.

When you drop a partitioned index, all partitions are dropped. When you drop a composite-partitioned index, all index partitions and subpartitions are dropped. When you drop a domain index, any statistics associated with the domain index are removed and any statistic types are disassociated. The optional keyword *FORCE* applies only when dropping domain indexes. *FORCE* allows you to drop a domain index marked *IN PROGRESS*, or to drop a domain index when its *indextype* routine invocation returns an error. For example:

```
DROP INDEX ndx_sales_salesperson_quota;
```

INDEXTYPE *indextype_name* [FORCE]

Drops the named *indextype* along with any associated statistic types. Oracle also drops any statistics collected with the statistics type. The optional keyword *FORCE* drops the *indextype* even when it is currently referenced by one or more domain indexes and marks the applicable domain indexes as *INVALID*. If you do not use *FORCE* on an *indextype* that is referenced by one or more domain indexes, the statement will fail. For example:

```
DROP INDEXTYPE imageindextype FORCE;
```

JAVA {SOURCE | CLASS | RESOURCE} *"java_object_name"*

Drops the named Java source, Java class, or Java resource schema object. *SOURCE* drops the named Java source schema object and all Java class schema objects derived from it. *CLASS* drops the named Java class schema object. *RESOURCE* drops the named Java resource schema object. For example, to drop the Java source *MyJavaSource* and all dependent schema objects:

```
DROP JAVA SOURCE "MyJavaSource";
```

LIBRARY *library_name*

Drops the named external procedure library.

{MATERIALIZED VIEW | SNAPSHOT} *mview_name*

Drops the named materialized view. You can use the keyword *SNAPSHOT* as an alternative to *MATERIALIZED VIEW* for backward compatibility. You must have privileges to drop the table plus views and indexes that make up the materialized view. Note that if you drop a parent table, any materialized views built upon it are not automatically dropped. But Oracle will return an error when a refresh is attempted on a materialized view whose parent table has been dropped. For example, we can drop a materialized view called **salespeople_at_quota**:

```
DROP MATERIALIZED VIEW salespeople_at_quota;
```

{MATERIALIZED VIEW | SNAPSHOT} LOG ON *parent_table_name*

Drops the named materialized view log that resides on the parent table (also known as the master table). You can use the statement *DROP SNAPSHOT* as an alternative to *DROP MATERIALIZED VIEW* for backward compatibility. RowID, primary key, and subquery materialized views can no longer be *fast refreshed* after the materialized view log has been dropped.

OPERATOR *operator_name* FORCE

Drops the named user-defined operator from the database. The optional keyword *FORCE* allows you to drop the operator even if it is referenced by other schema objects such as triggers, packages, user-defined functions, and so forth. In that case, the dependent objects become invalid. Dropping a user-defined operator with dependents without the *FORCE* keyword will result in an error.

OUTLINE *outline_name*

Drops the named stored outline from the database. The optimizer will generate new query plans for any SQL statements that used the stored outline after it is dropped.

PACKAGE [BODY] *package_name*

Drops the named package body and specification. The optional keyword *BODY* drops only the body of the package, but leaves the specification unaltered. This option is useful if you do not want to invalidate dependent objects, though you cannot call any components of the package until the package body is recreated. Otherwise, if you drop both body and specification, Oracle invalidates all dependent objects. (Note that if you want to drop a single object from a package, you should use the statement *CREATE PACKAGE [BODY]...OR REPLACE*.) The following examples drop first the package body and specification of the **sales_quota_calc** package, and then drop only the body of the **regional_sales_calc** package:

```
DROP PACKAGE sales_quota_calc;
```

```
DROP PACKAGE BODY regional_sales_calc;
```

PROCEDURE *procedure_name*

Drops the named stored procedure. Any dependent objects are invalidated when you drop a stored procedure. Dependent objects will fail with an error if they are accessed before you recreate the stored procedure. If you recreate the stored procedure, and a dependent object is accessed, the dependent object will be recompiled.

PROFILE *profile_name* [CASCADE]

Drops the named profile. The optional keyword *CASCADE* removes the profile from any users who were granted the profile. *CASCADE* is only required if the profile has been granted to one or more users. Users who no longer have a profile are then assigned the DEFAULT password. For example, the following statement drops the **web_user** profile and removes it from all users:

```
DROP PROFILE web_user CASCADE;
```

ROLE *role_name*

Drops the named role, removes it from the database, and revokes it from all users and roles where it has been granted. No new sessions can use the role, but sessions that are currently running under the role are not affected. For example, the following statement drops the **sales_mgr** role:

```
DROP ROLE sales_mgr;
```

ROLLBACK SEGMENT *rbseg_name*

Drops the named rollback segment and returns its space to the tablespace where the rollback segment was assigned. The rollback segment must be on-line and may not be the SYSTEM rollback segment. (Note that in Automatic Undo Management mode, you can only drop rollback segments. You cannot create or alter them.) For example, the following statement drops the rollback segment named **rbseg_sales_tbspc**:

```
DROP ROLLBACK SEGMENT rbseg_sales_tbspc;
```

SEQUENCE *sequence_name*

Drops the named sequence. (Note that the only way to restart a sequence is to drop it and recreate it.)

[PUBLIC] SYNONYM *synonym_name* [FORCE]

Drops the named private or *PUBLIC* synonym. (Note that the only way to redefine a synonym is to drop it and recreate it.) You cannot specify a schema when dropping a *PUBLIC* synonym. Materialized views become invalid when you drop their synonym. The optional keyword *FORCE* allows you to drop a synonym even if it has dependent tables or user-defined types. Otherwise, you cannot drop a synonym with dependent tables or user-defined types. Oracle discourages the use of *FORCE*. For example, you could drop the public synonym called **users**:

```
DROP PUBLIC SYNONYM users;
```

TABLE *table_name* [CASCADE CONSTRAINTS] [PURGE]

Drops the named table, erases all of its data, drops all indexes and triggers built from the table (even those in other schemas), and invalidates all permissions and all dependent objects (like views, stored procedures, etc.). On partitioned tables, Oracle drops all their partitions (and subpartitions). On index-organized tables, Oracle drops all dependent mapping tables. Statistic types associated with a dropped table are disassociated. Materialized view logs built on a table are also dropped when the table is dropped.

The *DROP TABLE* statement is effective for standard tables, index-organized tables, and object tables. The table being dropped is only moved to the recycling bin, unless you add the optional keyword *PURGE*, which tells Oracle to immediately free all space consumed by the table. (Oracle also supports a non-ANSI SQL command called *PURGE* that lets you remove tables from the recycling bin outside of the *DROP TABLE* statement.) *DROP TABLE* erases only the metadata of an external table. You must use an external operating system command to drop the file associated with an external table and reclaim its space.

Use the optional *CASCADE CONSTRAINTS* clause to drop all referential integrity constraints elsewhere in the database that depend on the primary or unique key of the dropped table. You cannot drop a table with dependent referential integrity constraints without using the *CASCADE CONSTRAINTS* clause. The following example drops the **job_desc**

table in the **emp** schema, then drops the **job** table and all referential integrity constraints that depend on the primary key and unique key of the **job** table:

```
DROP TABLE emp.job_desc;
```

```
DROP TABLE job CASCADE CONSTRAINTS;
```

TABLESPACE *tablespace_name* [INCLUDING CONTENTS [AND DATAFILES] [CASCADE CONSTRAINTS]

Drops the named tablespace (preferably when it is offline), deleting all datafiles and tempfiles. Oracle-managed datafiles and tempfiles in the tablespace are deleted from the filesystem, but datafiles and tempfiles managed by the OS can also be deleted by including the optional clause *INCLUDING CONTENTS AND DATAFILES*. Note that you cannot drop a tablespace when it contains a domain index, objects created by a domain index, data needed to roll back uncommitted transactions, or the *SYSTEM* tablespace. The optional clause *INCLUDING CONTENTS* drops all contents of the tablespace, and is required for any tablespace that contains database objects. The optional clause *AND DATAFILES* also drops associated operating system files. The optional clause *CASCADE CONSTRAINTS* drops all external referential integrity constraints that depend on the primary or unique keys of tables within the tablespace and is required when such dependencies exist. The first of the following examples drops the tablespace **sales_tbspc** and all dependent referential integrity constraints, while the second example drops the tablespace **sales_images_tbspc** and all associated operating system files:

```
DROP TABLESPACE sales_tbspc
```

```
INCLUDING CONTENTS CASCADE CONSTRAINTS;
```

```
DROP TABLESPACE sales_image_tbspc
```

```
INCLUDING CONTENTS AND DATAFILES;
```

TRIGGER *trigger_name*

Drops the named trigger from the database.

TYPE [BODY] *type_name* [{FORCE | VALIDATE}]

Drops the specification and body of the named object type, nested table type, or *VARRAY*, as long as they have no type or table dependencies. You must use the optional *FORCE* keyword to drop a supertype, a type with an associated statistics type, or a type with any sort of dependencies. All subtypes and statistic types are then invalidated. Oracle will also drop any public synonyms associated with a dropped type. The optional *BODY* keyword tells Oracle to drop only the body of the type while keeping its specification intact. Use the optional *VALIDATE* keyword when dropping subtypes to check for stored instances of the named type in any of its supertypes. Oracle then performs the drop only if no stored instances are found. For example:

```
DROP TYPE salesperson_type;
```

USER *user_name* [CASCADE]

Drops the named user from the database. By default, Oracle will not drop a user whose schema still owns objects. The optional keyword *CASCADE* tells Oracle to also remove all of the user's objects, as well as applicable dependent objects. Oracle does not drop objects from other users' schemas that reference those in the dropped schema, though it does invalidate them. For example, the following statement drops the user **cody** and all of his objects:

```
DROP USER cody CASCADE;
```

VIEW *view_name* [CASCADE CONSTRAINTS]

Drops the named view and marks as invalid any views, subviews, synonyms, or materialized views that refer to the dropped view. Use the optional clause *CASCADE CONSTRAINTS* to drop all referential integrity constraints that depend on the view. Otherwise, the *DROP* statement will fail if dependent referential integrity constraints exist. For example, the following statement drops the **active_employees** view in the **hr** schema:

```
DROP VIEW hr.active_employees;
```

In all of the syntax shown above, *object_name* can be replaced with [*schema_name.*]*object_name*. If you leave off the schema name, the default schema of the user session is assumed. Thus, the following *DROP* statement drops the specified view from the **sales_archive** schema:

```
DROP VIEW sales_archive.sales_1994;
```

However, if your personal schema is **scott**, then the following command is assumed to be against **scott.sales_1994**:

```
DROP VIEW sales_1994;
```

PostgreSQL

PostgreSQL does not support the *RESTRICT* or *CASCADE* optional keywords supported by the ANSI standard. It supports a wide variety of *DROP* variants, as follows:

```
DROP { AGGEGATE | DATABASE | FUNCTION | GROUP | INDEX |  
      LANGUAGE | OPERATOR | RULE | SEQUENCE | TABLE | TRIGGER |  
      TYPE | USER | VIEW } object_name
```

Following is the full syntax for each variant:

```
AGGEGATE aggregate_name ( { datatype | * } )
```

Drops the user-defined aggregate function. The value for *datatype* is the datatype of the existing aggregate function, but you may substitute an asterisk (***) to match any input datatype. For example, we can drop an aggregate function called **sales_avg** we made using PostgreSQL's *CREATE AGGREGATE* statement:

```
DROP AGGREGATE sales_avg(int4);
```

```
DATABASE database_name
```

Drops the named database and erases the operating system directory containing all of the database's data. This command can only be executed by the database owner while connected to a database other than the target database. For example, we can drop the **sales_archive** database:

```
DROP DATABASE sales_archive;
```

```
FUNCTION function_name ( [datatype1 [,...]] )
```

Drops the named user-defined function. Since PostgreSQL allows multiple functions of the same name distinguished only by the various input parameters they require, you must specify one or more datatypes to uniquely identify the user-defined function you wish to drop. PostgreSQL does not perform any kind of dependency checks on other objects that might reference a dropped user-defined function. (They will fail when invoked against an object that no longer exists.) For example:

```
DROP FUNCTION median_distribution (int, int, int, int);
```

```
GROUP group_name
```

Drops the named group, though it does not affect users who were in the group. This is similar to *DROP ROLE* on

other database platforms.

INDEX *index_name* [,...]

Drops one or more named indexes that you own. For example:

```
DROP INDEX ndx_titles, ndx_authors;
```

[PROCEDURAL] LANGUAGE *language_named*

Drops the named user-defined procedural language. You can optionally enclose the name in single quotes. Note that PostgreSQL does not check to see if any user-defined functions or triggers written in the named language still exist in the database. For example:

```
DROP LANGUAGE php;
```

OPERATOR *operator_id* ({*left_datatype* | NONE }, {*right_datatype* | NONE })

Drops a named user-defined operator where the value of *left_datatype* is the operator's left argument and *right_datatype* is the operator's right argument. *NONE* may be substituted for either argument when there is no argument in that position. (PostgreSQL does not check for any dependent access methods or operator classes that rely on the dropped operator. You should check for them and drop them manually to avoid errors when they are invoked.) For example, the first of the following statements drops the power operator, and the second drops the unary negation operator:

```
DROP OPERATOR ^ (int4, int4);
```

```
DROP OPERATOR ! (NONE, bool);
```

RULE *rule_name* [,...]

Drops one or more named rules from the database. PostgreSQL then immediately stops enforcing the rules and erases historical data about the rules. For example:

```
DROP RULE phone_rule;
```

SEQUENCE *seq_name* [,...]

Drops one or more named sequences from the database. Since the current version of PostgreSQL treats sequences as special tables, the *DROP SEQUENCE* statement works very much like the *DROP TABLE* statement.

TABLE *table_name* [,...]

Drops one or more existing tables from the database, as well as any indexes or triggers specified for the table. For example, you can drop more than one table in a single statement:

```
DROP TABLE authors, titles;
```

TRIGGER *trigger_name* ON *table_name*

Drops the named trigger from the database. Since PostgreSQL requires that trigger names be unique only on the table they are attached to, it is possible to have many triggers called, say, **insert_trigger** or **delete_trigger**, each on a different table. For example:

```
DROP TRIGGER insert_trigger ON authors;
```

TYPE *type_name* [,...]

Drops one or more pre-existing user-defined types from the database. PostgreSQL does not check to see what impact the *DROP TYPE* command might have on any dependent objects like functions, aggregates, or tables. You must check dependent objects manually. (Do *not* remove any of the built-in types that ship with PostgreSQL!) Note that PostgreSQL's implementation of types differs from the ANSI standard. Refer to the [CREATE/ALTER TYPE Statement](#) section for more information.

USER *user_name*

Drops the named user from the current database. This command does not drop any objects owned by the user, such as tables or views, but will fail if the user owns any databases. For example:

```
DROP USER emily;
```

VIEW *view_name* [,...]

Drops one or more pre-existing views from the database.

Note that PostgreSQL drop operations do not allow you to specify the target database where the operation will take place (except for *DROP DATABASE*). Therefore, you should execute any drop operation from the database where the object you want to drop is located.

SQL Server

SQL Server supports several standard variants of the *DROP* statement. SQL Server's main difference compared to the ANSI *DROP* statement is the ability to drop multiple objects with a single statement:

```
DROP { DATABASE | DEFAULT | FUNCTION | INDEX | PROCEDURE |  
      RULE | STATISTICS | TABLE | TRIGGER | TYPE | VIEW } object_name
```

Following is the full syntax for each variant:

DATABASE *database_name* [,...]

Drops the named database(s) and erases all disk files used by the database(s). This command may only be issued from the **master** database. Replicated databases must be removed from their replication schemes before they can be dropped, as must log shipping databases. You cannot drop a database while it is in use, nor can you drop system databases (**master**, **model**, **msdb**, or **tempdb**). For example, we can drop the **northwind** and **pubs** databases with one command:

```
DROP DATABASE northwind, pubs
```

```
GO
```

DEFAULT *default_name* [,...]

Drops one or more user-defined defaults from the current database. Note that this does not drop *DEFAULT* constraints, only user-defined defaults created with the *CREATE DEFAULT* statement. (You can drop default constraints using the statement *ALTER TABLE...DROP CONSTRAINT*.) When dropping a default from a column, null values will be inserted into columns that accept null. Otherwise, if a column does not accept nulls, an error is raised and no values are entered into the column. If a column or user-defined datatype being dropped is bound, then you must first unbind it using the SQL Server system stored procedure **sp_unbinddefault**. In the following two examples, we first drop a user-defined default and then unbind and drop the user-defined default:


```
DROP DEFAULT sales_date_default
```

```
GO
```

```
EXEC sp_unbindefault
```

```
'sales.ord_date'
```

```
DROP DEFAULT order_date_default
```

```
GO
```

```
FUNCTION [schema.]function_name [,...]
```

Drops one or more user-defined functions from the current database.

```
INDEX table_or_view_name.index_name [,...]
```

Drops one or more indexes from tables or indexed views in the current database and returns space to the database. This statement should not be used to drop a *PRIMARY KEY* or *UNIQUE* constraint. Instead, drop these constraints using the *ALTER TABLE...DROP CONSTRAINT* statement. When dropping a clustered index from a table, all non-clustered indexes are rebuilt. When dropping a clustered index from a view, all non-clustered indexes are dropped.

```
PROCEDURE procedure_name [,...]
```

Drops one or more stored procedures from the current database. SQL Server allows multiple versions of a single procedure via a version number, but these versions cannot be dropped individually. You must drop all versions of a stored procedure at once. System procedures (those with an **sp_** prefix) are dropped from the **master** database if they are not found in the current user database. For example:

```
DROP PROCEDURE calc_sales_quota
```

```
GO
```

```
RULE rule_name [,...]
```

Drops one or more user-defined rules from the current database. Existing data in columns governed by a rule are not checked when the rule is dropped. You may need to unbind the rule first using the SQL Server system stored procedure **sp_unbindefault**. In the following two examples, we first drop a user-defined rule, and then unbind and drop the user-defined rule:

```
DROP RULE phone_rule
```

```
GO
```

```
EXEC sp_unbindrule 'authors.phone'
```

```
DROP RULE phone_rule
```

```
GO
```

```
STATISTICS table_or_view_name.statistics_name [,...]
```

Drops statistics on one or more collections from a table or indexed view in the current database. Dropping statistics can have a big impact on query performance, since they help the query optimizer choose the best query plan. (Statistics are created with the *CREATE INDEX* and *CREATE STATISTICS* statements. You can update statistics with the *UPDATE STATISTICS* statement or retrieve information about them with the statement *DBCC SHOW_STATISTICS*.) For example, we'll create statistics on two columns in two separate tables and then

drop the statistics:

```
CREATE STATISTICS title_stats ON sales (title_id, ord_num)

WITH SAMPLE 30 PERCENT

GO

CREATE STATISTICS zip_stats ON stores (zip)

WITH FULLSCAN

GO

DROP STATISTICS sales.title_stats, stores.zip_stats

GO
```

TABLE *table_name*

Drops a named table and all data, permissions, indexes, triggers, and constraints specific to that table. (The *table_name* may be a fully qualified table name like **pubs.dbo.sales** or a simple table name, like **sales**, if the current database and owner are correct.) Views, functions, and stored procedures that reference the table are not dropped or marked as invalid, but will return an error when their procedural code encounters the missing table. Be sure to drop these yourself! You cannot drop a table referenced by a *FOREIGN KEY* constraint without first dropping the constraint. Similarly, you cannot drop a table used in replication without first removing it from the replication scheme. Any user-defined rules or defaults are unbound when the table is dropped. They must be rebound if the table is recreated.

TRIGGER *trigger_name* [,...]

Drops one or more triggers from the current database.

TYPE *type_name* [,...]

Drops one or more user-defined types from the current database.

VIEW *view_name* [,...]

Drops one or more views from the database, including indexed views, and returns all space to the database.

See Also

- CALL
- CONSTRAINTS
- CREATE/ALTER FUNCTION/PROCEDURE/METHOD
- CREATE SCHEMA
- CREATE/ALTER TABLE
- CREATE/ALTER VIEW
- DELETE
- DROP
- GRANT
- INSERT
- RETURN
- SELECT
- SUBQUERY
- UPDATE

EXCEPT Set Operator

The *EXCEPT* set operator retrieves the result sets of two or more queries including all the records retrieved by the first query that are not also found in subsequent queries. Whereas *JOIN* clauses are used to return the rows of two or more queries that are in common, *EXCEPT* is used to filter out the records that are present in only one of multiple, but similar, tables.

EXCEPT is in a class of keywords called a *set operator*. Other set operators include *INTERSECT* and *UNION*. (*MINUS* is Oracle's equivalent to the *EXCEPT* keyword. *EXCEPT* is the ANSI standard.) All set operators are used to simultaneously manipulate the result set of two or more queries, hence the term "set operators."

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Not supported

SQL2003 Syntax

There are technically no limits to the number of queries that you may combine with the *EXCEPT* operator. The general syntax is:

```
{SELECT statement1 | VALUES (expr1 [,...]) }
```

```
EXCEPT [ALL | DISTINCT]
```

```
[CORRESPONDING [BY (column1, column2, ...)] ]
```

```
{SELECT statement2 | VALUES (expr2 [,...]) }
```

```
EXCEPT [ALL | DISTINCT]
```

```
[CORRESPONDING [BY (column1, column2, ...)] ]
```

...

Keywords

VALUES (*expr1* [,...])

Generates a derived result set with explicitly declared values as *expression1*, *expression2*, etc. It is essentially a *SELECT* statement result set without the *SELECT...FROM* syntax. This is known as a *row constructor*, since the rows of the result set are manually constructed. According to the ANSI standard, multiple, hand-coded rows in a row constructor must be enclosed in parentheses and separated by commas.

EXCEPT

Determines which rows will be excluded from the single result set.

ALL

Considers duplicate rows from all result sets.

DISTINCT

Drops duplicate rows from all result sets prior to the *EXCEPT* comparison. Columns containing a NULL value are considered duplicates. (If neither *ALL* nor *DISTINCT* are used, *DISTINCT* behavior is the default.)

CORRESPONDING [BY] (*column1*, *column2*, ...)

Specifies that only columns with the listed names are returned, even if one or both queries use the asterisk shortcut.

Rules at a Glance

There is only one significant rule to remember when using *EXCEPT*:

The order, number, and datatypes of columns should be of the same category in all queries

The datatypes do not have to be identical, but they must be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest datatype size of each column in each ordinal position. For example, a query retrieving from columns containing a *VARCHAR(10)* and a *VARCHAR(15)* column will use the *VARCHAR(15)* datatype and size.

Programming Tips and Gotchas

None of the platforms support the *CORRESPONDING [BY column1, column2, ...]* clause.



On platforms that do not support *EXCEPT*, you might substitute a *NOT IN* subquery. However, *NOT IN* subqueries have different NULL handling and, on some database platforms, produce different result sets.

According to the ANSI standard, the *UNION* and *EXCEPT* set operators evaluate with equal precedence. However, the *INTERSECT* set operator evaluates first before the other set operators. We recommend that you explicitly control the precedence of set operators, using parentheses as a general best practice.

According to the ANSI standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. In order to avoid column and table ambiguity, be sure to alias each column for each table with the same respective alias. For example:

```
SELECT au_lname AS 'lastname', au_fname AS 'firstname'
FROM authors
EXCEPT
SELECT emp_lname AS 'lastname', emp_fname AS 'firstname'
FROM employees
ORDER BY lastname, firstname
```

Also, while each of your column lists may list columns with correspondingly compatible datatypes, there may be some variation in behavior across the DBMS platforms with regard to varying length of the columns. For example, if the **au_lname** column in the previous example's first query is markedly longer than the **emp_lname** column in the second query, each platform may apply different rules as to which length is used for the final result. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the names vary across column lists. In general, the column names of the first query are used.

DB2

DB2 supports the *EXCEPT* and *EXCEPT ALL* keywords of the ANSI standard, plus the *VALUES* extension:

```
{ SELECT statement1 | VALUES (expression1, expression2[,...]) }
EXCEPT [ALL]
{ SELECT statement2 | VALUES (expression1, expression2[,...]) }
EXCEPT [ALL]
...
```

where:

VALUES

Allows you to specify one or more hand-coded rows to include in the final result set. (This is known as a *row constructor*.) The *VALUES* clause must specify the exact same number of columns as the queries in the *EXCEPT*.

Although *EXCEPT DISTINCT* is not supported, *EXCEPT* is the functional equivalent. The *CORRESPONDING* clause is not supported. In addition, the datatypes *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK*, and the structured types are not usable with *EXCEPT*, but are useable with *EXCEPT ALL*.

When the result set has the same column name for a given column across all *SELECT* statements, then that name is used as the final column name returned by the statement. If, however, the given column is named differently in different *SELECT* statements, then you must rename the column in each query using the same *AS alias_name*.

If multiple set operations are used in a single query, those enclosed in parentheses are evaluated first. Then, the queries are evaluated in left-to-right order. However, all *INTERSECT* operations are performed before any *UNION* and *EXCEPT* operations.

For example:

```
(SELECT empno
  FROM employee
  WHERE workdept LIKE 'E%'
EXCEPT
SELECT empno
  FROM emp_act
  WHERE projno IN ('IF1000', 'IF2000', 'AD3110') )
UNION
VALUES ('AA0001'), ('AB0002'), ('AC0003')
```

The example above retrieves all IDs for all employees from the **employee** table who are in any department that starts with "E," then it excludes any employee IDs from the employee accounts (**emp_act**) table representing employees working in the projects **IF1000**, **IF200**, and **AD3110**. Finally, it adds in three extra employee IDs — **AA0001**, **AB0002**, and **AC0003** using the *UNION* set operator.

MySQL

EXCEPT is not supported in MySQL. However, you can use the *NOT IN* or *NOT EXISTS* operations as alternatives.

Oracle

Oracle does not support the *EXCEPT* set operator. However, it has an alternative set operator, *MINUS*, with identical functionality.

```
<SELECT statement1>
MINUS
<SELECT statement2>
MINUS
...
```

MINUS DISTINCT and *MINUS ALL* are not supported. *MINUS* is the functional equivalent of *MINUS DISTINCT*. Oracle does not support *MINUS* on queries under the following circumstances:

- Queries containing columns of *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatypes
- Queries containing the *FOR UPDATE* clause

- Queries containing *TABLE* collection expressions

If the first query in a set operation contains any expressions in the select item list, then include *AS* clauses to associate aliases with those expressions. Also, only the last query in the set operation may contain an *ORDER BY* clause.

For example, you could generate a list of all store IDs that do not have any records in the **sales** table:

```
SELECT stor_id FROM stores  
  
MINUS  
  
SELECT stor_id FROM sales
```

The *MINUS* command is functionally similar to a *NOT IN* query. The following query would retrieve the same results as the earlier query:

```
SELECT stor_id FROM stores  
  
WHERE stor_id NOT IN  
  
    (SELECT stor_id FROM sales)
```

PostgreSQL

PostgreSQL supports the *EXCEPT* and *EXCEPT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>  
  
EXCEPT [ALL]  
  
<SELECT statement2>  
  
EXCEPT [ALL]  
  
...
```

PostgreSQL does not support *EXCEPT* or *EXCEPT ALL* on queries with a *FOR UPDATE* clause. PostgreSQL does not support the *CORRESPONDING* clause. *EXCEPT DISTINCT* is not supported, but *EXCEPT* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *EXCEPT* or *EXCEPT ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the right-most occurrence of *ORDER BY* or *LIMIT* will be applied to the entire set operation.

PostgreSQL evaluates *SELECT* statements in a multi-*EXCEPT* statement from top to bottom unless you use parentheses to change the evaluation hierarchy of the statements.

Normally, duplicate rows are eliminated from the two result sets, unless you add the *ALL* keyword. For example, you could find all titles in the **authors** table that have no records in the **sales** table:

```
SELECT title_id  
  
FROM authors  
  
EXCEPT ALL  
  
SELECT title_id  
  
FROM sales;
```

SQL Server

EXCEPT is not supported. However, you can use *NOT IN* or *NOT EXISTS* operations in conjunction with a correlated subquery as alternatives. The following queries are examples of how you can achieve *EXCEPT* functionality using *NOT EXISTS* and *NOT IN*:

```
SELECT DISTINCT a.city
FROM pubs..authors AS a
WHERE NOT EXISTS
  (SELECT *
   FROM pubs..publishers AS p
   WHERE a.city = p.city)
SELECT DISTINCT a.city
FROM pubs..authors AS a
WHERE a.city NOT IN
  (SELECT p.city
   FROM pubs..publishers AS p
   WHERE p.city IS NOT NULL)
```

In general, *NOT EXISTS* is faster than *NOT IN*. In addition, there is a subtle issue with NULLs that differentiates the *IN* and *NOT IN* operators and the *EXISTS* and *NOT EXISTS* set operator. In order to get around this different handling of NULLs, simply add the *IS NOT NULL* clause to the *WHERE* clause, as we have in the preceding example.

See Also

```
INTERSECT
SELECT
UNION
```

EXISTS Operator

The *EXISTS* operator tests a subquery for the existence of rows.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
SELECT ...
WHERE [NOT] EXISTS (subquery)
```

The parameters and keywords are as follows:

WHERE [NOT] EXISTS

Tests the subquery for the existence of one or more rows. If even one row satisfies the subquery clause, it returns a Boolean TRUE value. The optional *NOT* keyword returns a Boolean TRUE value when the subquery

returns no matching rows.

subquery

Retrieves a result set based on a fully formed subquery.

Rules at a Glance

The *EXISTS* operator checks a subquery for the existence of one or more records against the records in the parent query.

For example, we want to see if we have any jobs where no employee is filling the position:

```
SELECT *  
FROM jobs  
WHERE NOT EXISTS  
    (SELECT * FROM employee  
     WHERE jobs.job_id = employee.job_id)
```

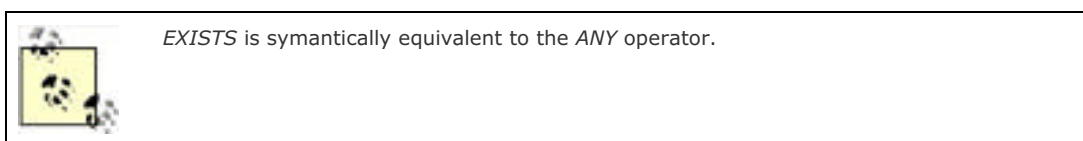
This example tests for the absence of records in the subquery using the optional *NOT* keyword. The next example looks for specific records in the subquery to retrieve the main result set:

```
SELECT au_lname  
FROM authors  
WHERE EXISTS  
    (SELECT *  
     FROM publishers  
     WHERE authors.city = publishers.city)
```

This query returns the last name of authors who live in the same city as their publishers. Note that the asterisk in the subquery is acceptable, since the subquery only needs to return a single record to provide a Boolean TRUE value. Columns are irrelevant in these cases. The first example selects only a single column. The key point is whether a *row* exists.

Programming Tips and Gotchas

EXISTS, in many queries, does the same thing as *ANY*. *EXISTS* is usually most effective with correlated subqueries.



The *EXISTS* subquery usually searches for only one of two things. Your first option is to use the asterisk wildcard (e.g., *SELECT * FROM...*) so that you are not retrieving any specific column or value. In this case, the asterisk means "any column." The second option is to select only a single column in the subquery (e.g., *SELECT au_id FROM...*). Some individual database platforms allow a subquery against more than one column (e.g., *SELECT au_id, au_lname FROM...*). However, this feature is rare and should be avoided on code that needs to be transportable across platforms.

Platform Differences

All of the platforms support *EXISTS* in the manner described above.

See Also

ALL/ANY/SOME
SELECT
WHERE

***FETCH* Statement**

The *FETCH* statement is one of four commands used in cursor processing, along with *DECLARE*, *OPEN*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than as a complete set. *FETCH* positions a cursor on a specific row and retrieves that row from the result set.

Cursors are especially important in relational databases because they are set-based, while most client-centric programming languages are row-based. So cursors allow you to perform operations a single row at a time, to better fit what a client program can do, rather than operate on a whole set of records at once.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |  
        { ABSOLUTE | RELATIVE int } }  
FROM ] cursor_name  
[INTO variable1 [, ...] ]
```

Keywords

NEXT

Tells the cursor to return the record immediately following the current row, and increments the current row to the row returned. *FETCH NEXT* is the default behavior for *FETCH*. It retrieves the first record if *FETCH* is performed as the first fetch against a cursor.

PRIOR

Tells the cursor to return the record immediately preceding the current row and decrements the current row to the row returned. *FETCH PRIOR* does not retrieve a record if it is performed as the first fetch against the cursor.

FIRST

Tells the cursor to return the first record in the cursor, making it the current row.

LAST

Tells the cursor to return the last record in the cursor, making it the current row.

ABSOLUTE int

Tells the cursor to return the *int* record from the cursor record set counting from the top (if *int* is a positive integer), or the *int* record counting from the bottom (if *int* is a negative integer), making the returned record the new current record of the cursor. If *int* is 0, no rows are returned. If the value of *int* moves the cursor past the end of the cursor result set, then the cursor is positioned after the last row (for a positive *int*) or before the first row (for a negative *int*).

RELATIVE int

Tells the cursor to return the record *int* rows after the current record (if *int* is positive) or *int* rows before the current record (if *int* is negative), making the returned record the new current row of the cursor. If *int* is 0, the current row is returned. If the value of *int* moves the cursor past the end of the cursor result set, then the cursor is positioned after the last row (for a positive *int*) or before the first row (for a negative *int*).

[FROM] cursor_name

Tells the name of the *OPEN* cursor where you want to retrieve rows. The cursor must be previously created and instantiated using the *DECLARE* and *OPEN* clauses. *FROM* is optional, but encouraged.

INTO variable1[, ...]

Stores data from each column in the open cursor into a local variable. Each column in the cursor must have a corresponding variable of a matching datatype in the *INTO* clause. Each column value is directly related to the variables in ordinal positions.

Rules at a Glance

At the highest level, a cursor must be:

1. Created using *DECLARE*
2. Opened using *OPEN*
3. Operated against using *FETCH*
4. Dismissed using *CLOSE*

By following these steps, you create a result set similar to that of a *SELECT* statement, except that you can operate against each individual row within the result set separately. For example, assume you've created and opened a cursor on a DB2 database called **employee_cursor** containing three columns:

```
DECLARE employee_cursor CURSOR FOR
```

```
SELECT lname, fname, emp_id
```

```
FROM employee
```

```
WHERE hire_date >= '2005-02-15';
```

```
OPEN employee_cursor;
```

You can now retrieve values from the cursor using *FETCH*:

```
FETCH FROM employee_cursor
```

```
INTO :emp_last_name, :emp_first_name, :emp_id
```

The *FETCH* statement allows you to position the current row within the cursor. Since you have not yet accessed the cursor in this example, you will be given the first record in the result set. You can then use any procedural code you want to operate against the values in that row.



Each database platform has its own rules about how variables are used. For example, DB2 requires a colon as a prefix on each variable, as shown in the earlier example; SQL Server requires an at sign (@), PostgreSQL and Oracle have no prefix, and so forth. Meanwhile, the SQL standard says that a colon prefix is necessary for languages that are embedded, like C or COBOL, but no prefix is needed for procedural SQL.

A cursor rests either directly on a row, before the first row, or after the last row. When a cursor is resting directly in a row, then that row is known as the *current row*. You can use cursors to position an *UPDATE* or *DELETE* statement in the current row using the *WHERE CURRENT OF* clause.

It is important to remember that a cursor result set does not wrap around. Thus if you have a result set containing 10 records and tell the cursor to move forward 12 records, it will not wrap back around to the beginning and continue its count from there. Instead, the default cursor behavior is to stop after the last record in the result set when scrolling forward, and to stop before the first record when scrolling back. For example, on SQL Server:

```
FETCH RELATIVE 12 FROM employee_cursor  
  
INTO @emp_last_name, @emp_first_name, @emp_id
```

When fetching values from the database into variables, make sure that the datatypes and number of columns and the variables match. Otherwise, you'll get an error. For example:

```
FETCH PRIOR FROM employee_cursor  
  
INTO @emp_last_name, @emp_id
```

The above example will fail since the **employee_cursor** contains three columns while the fetch operation has only two variables.

Programming Tips and Gotchas

The most common errors encountered with the *FETCH* statement are a mismatch of variables to the values in the cursor, whether that is in the exact number, order, or datatype. So take a little time to make sure you remember exactly what values are in the cursor and what their datatypes are.

Typically, the database will lock at least the current row and possibly all rows held by the cursor. According to the ANSI standard, cursor locks are not held through *ROLLBACK* or *COMMIT* operations, although this varies from platform to platform.

Although the *FETCH* statement is detailed in isolation here, it should always be managed as a group with the *DECLARE*, *OPEN*, and *CLOSE* statements. For example, every time you open a cursor, the server consumes memory. If you forget to close your cursors, you could create memory management problems. So make sure that every declared and opened cursor is eventually closed.

Cursors are also often used in stored procedures or batches of procedural code. The reason for this is because sometimes you need to perform actions on individual rows rather than on entire sets of data at a time. But because cursors operate on individual rows and not sets of data, they are often much slower than other means of accessing your data. Just make sure that you perform strong analysis on your approach. Many challenges, such as a convoluted *DELETE* operation or a very complex *UPDATE*, can be solved by using clever *WHERE* and *JOIN* clauses instead of cursors.

DB2

DB2 cursors are implicitly forward-only cursors that always scroll forward one record at a time. A DB2 cursor, when compared to the ANSI standard, is essentially a *FETCH NEXT 1* cursor. DB2 does not support keywords like *PRIOR*, *ABSOLUTE*, and *RELATIVE*. DB2 *FETCH* syntax is:

```
FETCH [ FROM ] cursor_name  
  
{ [INTO variable1 [, ...] ] |  
  
  [USING DESCRIPTOR descriptor_name] }
```

where:

USING DESCRIPTOR *descriptor_name*

Identifies a previously created SQLDA (SQL Descriptor Area) called *descriptor_name* to contain zero or more of the retrieved values from the *FETCH* statement.

When saving fetched values *INTO* variables, DB2 allows *LOB* values to be inserted into regular host variables such as

VARCHAR (when they are big enough to hold the value), locator variables, or file-reference variables.

When saving fetched values with the *USING DESCRIPTOR* clause, you should remember several rules about descriptor areas:

- *SQLN* indicates the number of *SQLVAR* occurrences in the *SQLDA*.
- *SQLDABC* indicates the number of bytes of storage allocated in the *SQLDA*.
- *SQLD* indicates the number of variables used in the *SQLDA* when processing the statement. *SQLD* must be set to a value greater than or equal to zero and less than or equal to *SQLN*.
- *SQLVAR* indicates the attributes of the variables. The descriptor needs enough storage to hold all *SQLVAR* occurrences. Use this formula to calculate the size of *SQLDABC*: it must be greater than or equal to $16 + \text{SQLN} * (n)$, where n is the length of *SQLVAR* occurrences.
- When *LOB* or structured type result columns must be accommodated, use two *SQLVAR* entries for every select item (or column returned from the result table).

For example, the following *FETCH* statement uses an *SQLDA*:

```
FETCH employee_cursor USING DESCRIPTOR :sqlda2
```

In the above example, the descriptor **:sqlda2** and the cursor **employee_cursor** must already exist.

MySQL

Not supported.

Oracle

Oracle cursors are implicitly forward-only cursors that always scroll forward one record at a time. An Oracle cursor, when compared to the ANSI standard, is essentially a *FETCH NEXT 1* cursor. Oracle cursors must either insert the retrieved values into matching variables, or use the *BULK COLLECT* clause to insert all of the records of the result set into an array. Oracle does not support keywords like *PRIOR*, *ABSOLUTE*, and *RELATIVE*. However, Oracle does support both forward-only and scrollable cursors in the database via the Oracle Call Interface (OCI). OCI also supports features like *PRIOR*, *ABSOLUTE*, and *RELATIVE* for read-only cursors whose result set is based on a read consistent snapshot.

Oracle's *FETCH* syntax is:

```
FETCH cursor_name
```

```
{ INTO variable_name1 [...] | BULK COLLECT INTO
```

```
  [collection_name [...] [LIMIT int] }
```

where:

```
BULK COLLECT INTO collection_name
```

Retrieves the entire rowset, or a specified number of rows (see *LIMIT*), into a client-side array or collection variable named *collection_name*.

```
LIMIT int
```

Limits the number of records fetched from the database when using the *BULK* statement. The *int* value is a nonzero integer (or a variable representing an integer value).

Oracle supports dynamic SQL-based cursors whose text can be built at runtime. Oracle also supports *cursor_names* that may be any allowable variable, parameter, or host-array type. In fact, you may also use user-defined or *%ROWTYPE* record variables for the *INTO* clause as well. This allows you to construct flexible, dynamic cursors in a template-like structure.

For example:

```
DECLARE

    TYPE namelist IS TABLE OF employee.lname%TYPE;

    names namelist;

    CURSOR employee_cursor IS SELECT lname FROM employee;

BEGIN

    OPEN employee_cursor;

    FETCH employee_cursor BULK COLLECT INTO names;

    ...

    CLOSE employee_cursor;

END;
```

Oracle *FETCH* often is paired with a PL/SQL *FOR* loop (or other kind of loop) to cycle through all the rows in the cursor. You should use the cursor attributes *%FOUND* or *%NOTFOUND* to detect the end of the rowset. For example:

```
DECLARE

    TYPE employee_cursor IS REF CURSOR RETURN employee%ROWTYPE;

    employee_cursor EmpCurTyp;

    employee_rec    employee%ROWTYPE;

BEGIN

    LOOP

        FETCH employee_cursor INTO employee_rec;

        EXIT WHEN employee_cursor%NOTFOUND;

        ...

    END LOOP;

CLOSE employee_cursor;

END;
```

This example uses a standard PL/SQL loop with an *EXIT* clause to end the loop when there are no more rows in the cursor to process.

PostgreSQL

PostgreSQL supports both forward and backward scrolling cursors. It covers the ANSI standard except *ABSOLUTE*, *FIRST*, or *LAST*. The syntax for *FETCH* in PostgreSQL is:

```
FETCH { [ FORWARD | BACKWARD | ABSOLUTE | RELATIVE ] }

    { [int | ALL | NEXT | PRIOR ] }

{ IN | FROM } cursor_name

[ INTO :variable1 [,...] ]
```

where:

FORWARD

Tells PostgreSQL that the cursor scrolls forward. This is the default if no other mode is defined.

BACKWARD

Tells PostgreSQL that the cursor scrolls backward.

ABSOLUTE

Allowed as a keyword, but not supported in functionality.

RELATIVE

Allowed as a keyword, but not supported in functionality.

int

A signed integer that indicates how many records to scroll forward (if positive) or backward (if negative).

ALL

Retrieves all records remaining in the cursor.

NEXT

Retrieves the next single record.

PRIOR

Retrieves the previous single record.

IN | FROM cursor_name

Defines the previously declared and opened cursor where data is retrieved.

INTO variable

Assigns a cursor value to a specific variable. As with ANSI cursors, the values retrieved by the cursor and the variables must match in number and datatype.

PostgreSQL cursors must be used with transactions explicitly declared using *BEGIN*, and must be closed with *COMMIT* or *ROLLBACK*.

The following PostgreSQL statement retrieves five records from the **employee** table and displays them:

```
FETCH FORWARD 5 IN employee_cursor;
```

It is important to note that PostgreSQL also supports a separate command, *MOVE*, to move to a specific cursor position. It differs from *FETCH* only by not returning values into variables:

```
MOVE { [ FORWARD | BACKWARD | ABSOLUTE | RELATIVE] }
```

```
{ [int | ALL | NEXT | PRIOR] }
```

```
{ IN | FROM } cursor_name
```

For example, the following code declares a cursor, skips forward five records, and then returns the values in the sixth record:

```
BEGIN WORK;  
  
    DECLARE employee_cursor CURSOR FOR SELECT * FROM employee;  
  
    MOVE FORWARD 5 IN employee_cursor;  
  
    FETCH 1 IN employee_cursor;  
  
    CLOSE employee_cursor;  
  
COMMIT WORK;
```

The above code will return a single row from the **employee_cursor** result set.

SQL Server

SQL Server supports a variation of *FETCH* that is very close to the ANSI standard:

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |  
        { ABSOLUTE | RELATIVE int } }  
[FROM] [GLOBAL] cursor_name  
[INTO @variable1 [, ...] ]
```

The differences between SQL Server's implementation and the ANSI standard are very small. First, SQL Server allows you to use variables in place of the *int* and *cursor_name* values. Second, SQL Server allows the declaration and use of *GLOBAL* cursors that can be accessed by any user or session, not just the one that created it.

There are some rules that apply to how you use the *FETCH* command based upon how you issued the *DECLARE CURSOR* command:

- When you declare a *SCROLL* SQL-92 cursor, all *FETCH* options are supported. In all other SQL-92 cursors, only *NEXT* is the only option supported. (There is also an alternate Transact-SQL style *DECLARE CURSOR* statement.)
- *DYNAMIC SCROLL* cursors support all *FETCH* options except *ABSOLUTE*.
- *FORWARD-ONLY* and *FAST_FORWARD* only support *FETCH NEXT*.
- *KEYSET*, *STATIC*, and *SCROLL* cursors (and *DYNAMIC*, *FORWARD-ONLY*, and *FAST_FORWARD* are not defined), all *FETCH* options are supported.



SQL Server also requires a *DEALLOCATE* statement, in addition to *CLOSE*, to release memory consumed by a cursor.

Here's a full example that goes from declaring and opening a cursor to initiating several fetches and then finally closing and deallocating the cursor:

```
DECLARE @vc_lname VARCHAR(30), @vc_fname VARCHAR(30), @i_emp_id CHAR(5)  
  
DECLARE employee_cursor SCROLL CURSOR FOR  
  
    SELECT lname, fname, emp_id  
  
    FROM employee  
  
    WHERE hire_date <= 'FEB-14-2004'  
  
OPEN employee_cursor  
  
-- Fetch the last row in the cursor.
```

```
FETCH LAST FROM employee_cursor
-- Fetch the row immediately prior to the current row in the cursor.
FETCH PRIOR FROM employee_cursor
-- Fetch the fifth row in the cursor.
FETCH ABSOLUTE 5 FROM employee_cursor
-- Fetch the row that is two rows after the current row.
FETCH RELATIVE 2 FROM employee_cursor
-- Fetch values eight rows prior to the current row into variables.
FETCH RELATIVE -8 FROM employee_cursor
INTO @vc_lname, @vc_fname, @i_emp_id
CLOSE employee_cursor
DEALLOCATE employee_cursor
GO
```

Remember that in SQL Server you must not only *CLOSE* the cursor, but you must also *DEALLOCATE* it. In some rare cases, you might wish to reopen a closed cursor. You can reuse any cursor that you have closed but not deallocated. The cursor is permanently destroyed only when it is deallocated.

See Also

- CLOSE
- DECLARE CURSOR
- OPEN

GRANT Statement

The *GRANT* statement assigns privileges to users and roles, which allows them to access and use database objects (that is, object privileges).

In addition, most database platforms use the *GRANT* statement to authorize users and roles to create database objects and execute stored procedures, functions, and so on (particularly usage privileges).

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 syntax for *GRANT* includes only the assignment of object privileges and roles to a specific user:

```
GRANT { {object_privilege [,...]} | role [,...]} }
```

```
[ON database_object_name]
```

```
[TO grantee [,...]]
```


[WITH HIERARCHY OPTION] [WITH GRANT OPTION] [WITH ADMIN OPTION]

[FROM { CURRENT_USER | CURRENT_ROLE }]

Keywords

object privilege

Grants one or more privileges for a SQL statement. Multiple privileges may be granted at one time, as long as commas separate them. Do not combine *ALL PRIVILEGES* with other privileges, but the remainder may be combined in any order. Valid privileges are:

ALL PRIVILEGES

Indicates a shorthand for every privilege the grantor has available to grant. Grants all privileges currently assigned to the grantor to the named users and/or for the specified database objects. This is generally not a recommended approach, since it can encourage sloppy permissioning.

EXECUTE

Grants the privileges to execute any ANSI SQL *routine*. An ANSI routine is any stored procedure, user-defined function, or user-defined method.

{SELECT | INSERT | UPDATE | DELETE}

Grants the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns of a given table on which the privilege will be granted (except for the *DELETE* statement).

REFERENCES

Grants the privilege to use the column(s) in any constraint or assertion, not just a foreign key. The *REFERENCES* clause also allows you to define column-specific details (by including a parenthetical, comma-delimited list) for statements like *INSERT*, *SELECT*, and so on. Finally, *REFERENCES* grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

TRIGGER

Grants the privilege of creating a trigger to operate on a specific table and its columns.

UNDER

Grants the privilege to create subtypes or typed tables.

USAGE

Grants privileges to use a domain, user-defined type, character set, collation, sequence, or translation.

role

Grants a specific, predefined role from the named grantee identified in the *FROM* clause. For example, the database administrator might want to create a role called **Reporter** that has read-only access to several tables. By granting this role to a user, that user then receives all of the read-only permissions granted to the **Reporter** role. Users can have a role assigned to them and receive all of the permissions already assigned to that role.

ON *database_object_name*

Grants privileges on the specified, predefined database object identified by *database_object_name*, which may be

one of the following:

```
{[TABLE] object_name | DOMAIN object_name |  
COLLATION object_name | CHARACTER SET object_name |  
TRANSLATION object_name | SPECIFIC ROUTINE routine_name}
```

TO *grantee*

Assigns the privilege to the named user(s) or role(s), designated as *grantee*. You can assign privileges to multiple users and/or roles, as long as commas separate them. Alternately, privileges may be granted to *PUBLIC*, meaning that all users (including those that will be created in the future) have the specified privileges.

WITH HIERARCHY OPTION

Bestows the *WITH HIERARCHY OPTION* privilege, enabling the grantee to *SELECT* not only from the named table, but also from all of its subtables. This applies to privilege grants only.

WITH GRANT OPTION

Enables the grantee to further grant the privileges to other users. This applies to privilege grants only.

WITH ADMIN OPTION

Bestows the ability to assign a role. The ability to assign role responsibilities is separate from the ability to grant privileges to other users or roles.

FROM { *CURRENT_USER* | *CURRENT_ROLE*}

Names the user who is granting the specified privilege(s), either the *CURRENT_USER* or the *CURRENT_ROLE*. This clause is optional and assumes the current user context.

Rules at a Glance

You may grant many privileges on a single object with one statement, but do not mix the *ALL PRIVILEGES* privilege with the individual SQL statement keywords in a single *GRANT* statement. You can grant a specific privilege on a specific database object to a single user using:

```
GRANT privilege_name ON object_name TO grantee_name
```

For example:

```
GRANT SELECT ON employee TO Dylan;
```

You can grant a specific privilege on a specific object to all users via the *PUBLIC* user. When a grant is issued to *PUBLIC*, that means everyone has permission to the object without the need for a grant specifically to the object. For example:

```
GRANT SELECT ON employee TO PUBLIC;
```

When granting privileges to multiple grantees, simply place a comma between each:

```
GRANT SELECT ON employee TO Dylan, Matt
```

The previous example shows that you may grant privileges to multiple users, in this case **Dylan** and **Matt**, and to the *PUBLIC* user in a single *GRANT* statement. When granting privileges on a table, the privileges may extend to the column level by showing a list of columns enclosed in parentheses after the table name. For example:

```
GRANT SELECT ON employee(emp_id, emp_fname, emp_lname, job_id)
```

```
TO Dylan, Matt
```

The previous example shows that **Dylan** and **Matt** now have select privileges on several columns of the **employee** table.

Programming Tips and Gotchas

Depending on the specific database implementation, views may or may not have independent access privileges from their base tables.

Note that all of the optional *WITH* clauses control extensions to the base privilege(s) granted in the statement. Thus, the following command grants the privilege to *SELECT* records from the **employee** table:

```
GRANT SELECT ON employee TO Dylan;
```

but the following command grants the privilege to *SELECT* records from the **employee** table *and* also to grant that *SELECT* privilege to other users.

```
GRANT SELECT ON employee TO Dylan
```

```
WITH GRANT OPTION;
```

Similarly, you can use the *REVOKE* statement to revoke only the *WITH GRANT OPTION*, the individual *SELECT* privilege, or both.

Most of the platforms isolate privileges at the user and role level. Thus, an individual user who is a member of two roles may have an individual set of permissions four times — once for themselves, once for *PUBLIC*, and once with each of the roles for which they have permission. In this situation, you should be very careful. In order to remove a specific privilege completely from such a user, you have to revoke the user's privileges directly, and then in both roles.

DB2

DB2 offers many options in its implementation of the *GRANT* command. It is important to note that privileges on database objects (such as *SELECT* privileges on a specific table) are handled by the command, as are a variety of other privileges:

```
GRANT { [ database_privilege [,...] ON DATABASE ] |  
  [ CONTROL ON INDEX index_name ] |  
  [ {BIND | CONTROL | EXECUTE} ON PACKAGE package_name ] |  
  [ EXECUTE ON [SPECIFIC] {FUNCTION | METHOD {* | FOR} | PROCEDURE}  
    object_name ] |  
  [ {ALTERIN | CREATEIN | DROPIN} ON SCHEMA schema_name ] |  
  [ ON SEQUENCE sequence_name ] |  
  [ PASSTHRU ON SERVER server_name ] |  
  [ object_privilege [,...] ON [TABLE] object_name ] }  
TO [ {USER |GROUP} ]  
  { grantee_name [,...] | group_name [,...] | PUBLIC } ]  
[WITH GRANT OPTION] | [ USE OF TABLESPACE tablespace_name ]
```

The parameters are as follows:

database_privileges

Grants one or more privileges that apply to the entire database, rather than to a single object. The database privileges are:

BINDADD

Grants the authority to create packages. The user who creates a package always has the *CONTROL* privilege on that package even when *BINDADD* is revoked at a later date.

CONNECT

Grants the authority to access the specified database.

CREATETAB

Grants the authority to create a base table. The user who created the base table always has the *CONTROL* privilege on the table even if *CREATETAB* is revoked at a later date.

CREATE_EXTERNAL_ROUTINE

Grants the privilege to register external routines. This is somewhat dangerous since external threads can have adverse side effects. The routine will continue to exist once created, even if the *CREATE_EXTERNAL_ROUTINE* privilege is revoked for the user. (For more information, check on the *THREADSAFE* clause of the *CREATE* or *ALTER* routine statements in the DB2 documentation.)

CREATE_NOT_FENCED_ROUTINE

Grants the authority to create routines that are not fenced into a specific user process. That is, the routines may execute within the database manager's process. *CREATE_NOT_FENCED_ROUTINE* is a superset of *CREATE_EXTERNAL_ROUTINE*. Consequently, you automatically are granted the latter privilege when assigned the former. Read the description of *CREATE_EXTERNAL_ROUTINE* for more information.

DBADM

Grants superuser authority, including the ability to create any object, and including all privileges against all objects in the database, even the ability to grant those privileges to other users.

IMPLICIT_SCHEMA

Grants the authority to create a schema implicitly, though this privilege does not impact the ability to create a schema or any objects within a schema explicitly.

LOAD

Grants the authority to explicitly *LOAD* the specified database. *SYSADM* and *DBADM*, who automatically possess this privilege, need no other explicitly granted privileges to load a database. But other users with *LOAD* privilege must have, at a minimum, *INSERT* privileges on tables they wish to load.

QUIESCE_CONNECT

Grants the privilege to access the specified database while it is quiesced.

CONTROL ON INDEX *index_name*

Grants the privilege to drop an index, once created, of the specified *index_name*. This is the explicit *CONTROL* privilege for indexes. It is automatically granted to the creator of any index.

{BIND | CONTROL | EXECUTE} ON PACKAGE *package_name*

Grants the privileges associated with packages. Note that *RUN* and *PROGRAM* are acceptable synonyms for *EXECUTE* and *PACKAGE*, respectively. The package privileges are:

BIND

Grants the authority to bind or rebind the specified package. Since any SQL statements in the package are checked at bind time, the user must also have statement permissions on each object referenced by the package.

CONTROL

Grants the authority to rebind, drop, or execute the specified package, as well as to grant *BIND* and *EXECUTE* privileges to other users. Users granted the *CONTROL* privilege automatically receive *BIND* and *EXECUTE* as well. *CONTROL* is automatically granted to the creator of a package.

EXECUTE

Grants the authority to execute the specified package.

EXECUTE ON [SPECIFIC] {FUNCTION | METHOD {* | FOR} | PROCEDURE} *object_name*

Grants privileges on an ANSI routine (i.e., user-defined function, user-defined method, or stored procedure). The elements of this clause are as follows:

FUNCTION [*schema.*]*function_name*

Specifies that the user is granted execution privileges on a specific function. A *SPECIFIC* function name may be specified. DB2 also supports functions without names. In this case, *FUNCTION [(param [...])]* should be used, including any and all parameters of the function. If *schema* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific function name to grant a user the execution privilege on all functions in the current or specified schema.

METHOD { * | FOR *type_name*}

Specifies that the user is granted rights to execute all methods of *type_name*, even those that have not yet been created. The optional *FOR type_name* clause names the type where the specified method is found. A *SPECIFIC* method name may be specified. DB2 also supports methods without names. In this case, *METHOD [(param [...])]* should be used, including any and all parameters of the function. Alternately, the asterisk (*) can be used to specify that all types in the schema are granted to the user.

PROCEDURE [*schema.*]*procedure_name*

Specifies that the user be granted execute privileges on all stored procedures in the specified schema, even those created in the future. If *schema* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific stored procedure name to grant a user the execution privilege on all stored procedures in the current or specified schema.

{ALTERIN | CREATEIN | DROPIN} ON SCHEMA *schema_name*

Grants privileges associated with manipulating a schema (that is, a user-owned set of objects). Only users with *SYSADM*, *DBADM*, or *WITH GRANT OPTION* permissions for all privileges for the specified schema may issue this command. The elements are as follows:

ALTERIN

Grants the authority to alter or comment on objects in a schema defined by *ON SCHEMA schema_name*.

CREATEIN

Grants the authority to create objects in a schema defined by *ON SCHEMA schema_name*.

DROPIN

Grants the authority to drop objects in the schema defined by *ON SCHEMA schema_name*.

{ALTER | USAGE} ON SEQUENCE *sequence_name* TO PUBLIC

Grants privileges on a sequence in an explicit or current schema (*USAGE*) or the privilege to execute the *ALTER SEQUENCE* statement (*ALTER*).

PASSTHRU ON SERVER *server_name*

Grants the privilege to access an external data source named *server_name* using pass-through mode.

object_privilege

Represents privileges for a variety of statements on tables, views, and nicknamed tables or views. Object privileges may be combined in any order:

ALL [PRIVILEGES]

Provides a shorthand for every privilege the grantor has available to grant. The *PRIVILEGES* keyword is optional. It grants all privileges currently assigned to the granting user for the specified database objects, including *SELECT*, *INSERT*, *UPDATE*, and *DELETE* privileges. With *ALL*, you have privileges to add columns to a table, create or drop primary keys or unique constraints, create or drop foreign keys, create or drop check constraints, create triggers on a table, add or change comments on a table, change a nicknamed column name or datatype, or add, rest, or drop a nicknamed column name. However, granting *ALL* is generally not a good approach since it can encourage sloppy coding.

ALTER

The privilege to add or alter columns, constraints, comments, or triggers on a specified table or view. *ALTER* also permits the use of nicknames on column names or datatypes or to change a column option for a nickname.

CONTROL

Grants a large number of privileges to the specific user or role (or *PUBLIC*) for the specified object. When you grant *CONTROL* on a table, that includes the privileges for *ALTER*, *DELETE*, *INSERT*, *INDEX*, *REFERENCES*, *SELECT*, and *UPDATE*. When you grant *CONTROL* on a view, that includes *DELETE*, *INSERT*, *SELECT*, and *UPDATE*. On nicknames, *CONTROL* includes *ALTER*, *INDEX*, and *REFERENCES*.

The grantee has the ability to grant these aforementioned privileges to others, who may drop the specified table, view, or nickname. They may execute the *RUNSTATS* utility on tables and indexes, as well as execute the *SET INTEGRITY* command on tables, materialized views, and staging tables.

INDEX

The ability to create an index against a table, or to create an index specification on a nickname. (The creator of an index retains *CONTROL* privileges on the indexes they create, even if the *INDEX* privilege is later revoked.)

{*SELECT* | *INSERT* | *UPDATE* | *DELETE*}

Grants the specific privilege to a given user for the specified database object, such as a table or view. In some cases, these privileges also extend to utilities with similar behavior. For example, the *INDEX* privilege also includes the use of the *IMPORT* utility. You may add a parenthetical list of the columns on a given table where the privilege is granted. The *UPDATE* privilege may be assigned down to the column level by including a parentheses-encapsulated list of column or columns made available to the user.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object. The *REFERENCES* privilege may be assigned down to the column level by including a parentheses-encapsulated list of column or columns made available to the grantee.

TO { [USER | GROUP]} *grantee_name*

Names the user or users who will gain the specified privilege or privileges. When both a user and group exist with the same name, use the optional *USER* or *GROUP* keywords to control whether the privileges are granted to the user or to the group. *PUBLIC* may also be used to grant privileges to the *PUBLIC* role. Multiple grantees can be listed; separate them with a comma between each.

WITH GRANT OPTION

Entitles the grantee to assign the granted privilege(s) to other users.

USE OF TABLESPACE *tablespace_name*

Grants the privilege of specifying or defaulting new tables to the specified *tablespace_name*. In effect, it enables the *USE* privilege on that tablespace.

In situations when a group and a user of the same name exist, you must use either *GROUP* or *KEYWORD* to inform DB2 whether you want the privileges assigned to the group or to the user. If you do not, DB2 will return an error and abort the *GRANT* statement. For example, to grant *CONNECT* and *BINDADD* permissions to the user **emily** and the group **datareaders**:

```
GRANT CONNECT, BINDADD ON DATABASE TO USER emily,  
GROUP datareaders
```

To give the user **Dylan**, and all users who are members of the **Admin** role, control over a specific index:

```
GRANT CONTROL ON INDEX emp_ndx TO dylan, admin;
```

You can give a user rights on a package to execute it and bind (or rebind it), without the privilege to drop it (via *CONTROL*) as follows:

```
GRANT BIND, EXECUTE ON PACKAGE d02.sales_processing  
TO USER dylan
```

The following example grants a user a variety of privileges on the **sales** table, as well as the privilege to pass those privileges on to other users:

```
GRANT DELETE, INSERT, SELECT, UPDATE ON sales TO USER sam  
WITH GRANT OPTION
```

Similarly, you can give another user privileges on a specific object (a view in this case) that includes only specific columns in the object:

```
GRANT UPDATE (sales_qty, sales_discount) ON sales TO USER jake
```



DB2 also has privileges at the instance level that aren't managed with the *GRANT* statement. Most importantly, there are three database manager configuration parameters (*SYSADM_GRP*, *SYSCTRL_GRP*, and *SYSMAINT_GRP*) that control administrative rights to the instance. You control these privileges using the DB2 *UPDATE DATABASE MANAGER CONFIGURATION* statement. For example:

```
ATTACH DB2;
```

```
UPDATE DATABASE MANAGER CONFIGURATION
  USING
    sysadm_group 'DB2GRP'
  IMMEDIATE;

DETACH;
```

MySQL

MySQL provides additional access privileges, primarily relating to object manipulation within a database. *GRANT* in MySQL is available from Version 3.22.11 and on. The syntax is as follows:

```
GRANT [ { ALL [PRIVILEGES] |
  {SELECT | INSERT | UPDATE} [ (column_name [...]) ] | DELETE |
  REFERENCES [ (column_name [...]) ] } |
[ USAGE ] | [{ALTER | CREATE | DROP}] | [FILE] | [INDEX] | [PROCESS] |
[RELOAD] | [SHUTDOWN] | [CREATE TEMPORARY TABLES] | [LOCK TABLES] |
[REPLICATION CLIENT] | [REPLICATION SLAVE] | [SHOW DATABASES] |
[SUPER] }[,...]
ON {table_name | * | *.* | database_name.*}
[TO grantee_name [IDENTIFIED BY 'password']] [,...]
[WITH GRANT OPTION]
```

where:

ALL [PRIVILEGES]

Grants a user all applicable privileges that apply at the current level on one or more objects in the database, with the exception of *WITH GRANT OPTION*. For example, *GRANT ALL ON *.** grants all privileges, except *WITH GRANT OPTION*, globally.

SELECT | INSERT | UPDATE | DELETE

Grants the ability to read, write, modify, and erase data from tables (and possibly from specific columns of a table), respectively. Column-level permissions are allowed on *SELECT*, *INSERT*, and *UPDATE*, but not *DELETE*. Granting column-level permissions can slow down MySQL somewhat, since more permission checking is required.

REFERENCES

Remains unimplemented.

USAGE

Creates a "no privilege" user account for use in creating a user that has no privileges.

ALTER | CREATE | DROP

Grants the ability to alter, drop, or create tables and other database objects. When using the *CREATE* syntax, you are not required to specify an *ON* clause to identify a specific table.

FILE

Grants the ability to load data from, or write data to, files using the *SELECT INTO* and *LOAD DATA* commands.

INDEX

Grants the ability to create or drop indexes.

PROCESS

Grants the ability to view running processes using *SHOW FULL PROCESSLIST*.

RELOAD

Grants the ability to invoke the *FLUSH* and *RESET* commands.

SHUTDOWN

Grants usage of the *mysqladmin shutdown* command to kill the server process.

CREATE TEMPORARY TABLES

Grants the ability to create temporary tables.

LOCK TABLES

Grants the use of the MySQL command *LOCK TABLES* on tables where the user has *SELECT* privileges.

REPLICATION CLIENT

Grants a user the privilege to see metadata about replication slaves and masters.

REPLICATION SLAVE

Grants a replication slave the right to read binlogs from a replication master.

SHOW DATABASES

Grants the user the privilege to execute the MySQL command *SHOW DATABASES*.

SUPER

Grants the user the privilege of one connection even if the *max_connections* threshold is reached. Users with *SUPER* privileges can also execute important MySQL commands like *CHANGE MASTER*, *KILL*, *mysqladmin debug*, *PURGE [MASTER] LOGS*, and *SET GLOBAL*.

*ON {table_name | * | *.* | database_name.*}*

Grants privileges from the specified *table_name*, from all tables within the current database using an asterisk (*), from all tables in all databases with **.**, or from all tables in the specified *database_name* with *database_name.**.

TO *grantee_name* [IDENTIFIED BY '*password*'] [,...]

Names the user or users who will gain the specified privilege. You may enter multiple users separated by commas. When granting privileges to a new user, you may also optionally set the new password for the new named user at the same time using the *IDENTIFIED BY* clause.

WITH GRANT OPTION

Entitles the grantee to assign the granted privilege to other users.

Since MySQL is focused on speed, you can implement server-wide features that provide high-speed performance. For example, you can enable the *SKIP_GRANT_TABLES* startup option to disable checks for privileges. This can speed up queries, but obviously no permission checking is done. This means that all users have full access to all resources in the database.

The following are access privileges that are usable with tables: *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE*, *DROP*, *GRANT*, *INDEX*, and *ALTER*. At the column level, you can apply *INSERT*, *UPDATE*, and *SELECT*.

In any situation, the host, table, database, and column names may each be up to 60 characters in size. The username (a.k.a. the *grantee_name*) may be up to 16 characters.

MySQL also supports the possibility of granting rights to a specific user on a specific host if the *grantee_name* is in the form *USER@HOST*. Wildcards can be included in the hostname of a *grantee_name* to provide the access privilege to a large number of users at one time. Also, a missing hostname is the same as the "%" wildcard. For example:

```
GRANT SELECT ON employee TO katie@% IDENTIFIED BY 'audi',  
    annalynn IDENTIFIED BY 'cadillac',  
    cody@us% IDENTIFIED BY 'bmw';
```

The preceding *GRANT* statement grants read privileges on the employee table to the user **katie** explicitly at any hostname. Similarly, the *GRANT* statements grants read privileges to the user **annalynn** implicitly at any hostname. And finally, the *GRANT* statement grants read privileges to the user **cody** explicitly at any hostname starting with "US".

The *grantee_name* must be 16 or fewer characters. When specifying the user, password protection may be enforced by including the *IDENTIFIED BY* clause.

The following example grants permissions to three users with passwords:

```
GRANT SELECT ON employee TO dylan IDENTIFIED BY 'porsche',  
    kelly IDENTIFIED BY 'mercedes',  
    emily IDENTIFIED BY 'saab';
```

In MySQL, if you grant permissions to a user that doesn't exist, it creates that user. The two statements that follow create a user and password with global privileges and another user that has no privileges (the *USAGE* privilege):

```
GRANT SELECT ON *.* TO tony IDENTIFIED BY 'humbolt';  
GRANT USAGE ON sales.* TO alicia IDENTIFIED BY 'dakota';
```



Failing to include the *IDENTIFIED BY* clause on a *GRANT* statement that creates a new user will create a user without a password. This practice is insecure and dangerous.

If you do create a new user using the *GRANT* command, and you forget to specify a password, you can, and probably should, set a password using the *SET PASSWORD* command.

MySQL also allows you to issue grants on remote hosts, specified with *@hostname*, and include the wildcard character percent (%). For example, the following grant provides all permissions on all tables of the current database across all hosts in the **notforprofit.org** domain to the user **annalynn**:

```
GRANT ALL ON * TO annalynn@%.notforprofit.org;
```

Privileges are written directly to system tables at four different levels:

Global level

Global privileges apply to all databases on a given server and are stored in the **mysql.user** system table.

Database level

Database privileges apply to all tables in a given database and are stored in the **mysql.db** system table.

Table level

Table privileges apply to all columns in a given table and are stored in the **mysql.tables_priv** system table.

Column level

Column privileges apply to single columns in a given table and are stored in the **mysql.columns_priv** system table.

Because of the usage and system tables involved, you may grant redundant permissions at different levels. For example, you could grant a user *SELECT* privileges on a specific table, and then grant global *SELECT* privileges on that same table to all users of the database. You would then have to revoke those privileges separately to ensure the user no longer had the privilege of *SELECT*ing from the table.

The system tables are also somewhat unprotected, so it is possible to change permissions by issuing *INSERT*, *UPDATE*, and *DELETE* statements against the tables rather than by using *GRANT* or *REVOKE*. Normally, all privileges are read into memory when MySQL starts. Database, table, and column privileges issued by a *GRANT* statement are available immediately and take effect right away. User privileges issued by a *GRANT* statement are noticed immediately and take effect the next time the user connects. If you directly modify permissions in the system tables, none of the changes are noticed until you restart MySQL, or until you issue a *FLUSH PRIVILEGES* command.

Oracle

Oracle's implementation of the *GRANT* statement supports an enormous number of variations and permutations. The syntax is as follows:

```
GRANT { [object_privilege] [,...] |  
        [system_privilege] [,...] |  
        [role] [,...] }  
[ ON { [schema_name.][object] [,...] |  
        [DIRECTORY directory_object_name] |  
        [JAVA { SOURCE | RESOURCE } ] [schema_name.][object] }  
TO { {grantee_name [,...] | role_name [,...] | PUBLIC}  
    [WITH { GRANT | HIERARCHY } OPTION]  
[IDENTIFIED BY password][WITH ADMIN OPTION];
```

You can grant multiple privileges with a single invocation of the command, but they must all be of the same type (object, system, or role).

For example, you could grant a user three object privileges on a single table in one *GRANT* statement, and then issue a separate grant of two system privileges to a role, and yet a third grant of several roles to a user, but not all of them in one *GRANT* statement.

Following are the parameters to Oracle's *GRANT* statement:

object_privilege

Grants the specified privileges to the specified *grantee_name* or role on the named schema object (for example, a table or view). You may combine several object privileges, schema objects, or grantees in a single statement. However, you may *not* grant system privileges or roles in the same statement as object privileges. The object privileges are:

ALL [PRIVILEGES]

Grants all available privileges on a named schema object and is available for use on tables.

ALTER

Grants the privilege to change an existing table using the *ALTER TABLE* statement, and is available for use on tables and sequences.

DEBUG

Grants access to a table through a debugger. The access granted applies to any triggers on the table or to information on SQL that directly references the table and is available for use on tables, views, procedures, functions, packages, Java objects, and types.

EXECUTE

Grants the privilege to execute a stored procedure, user-defined function, or a package, and is available for use on procedures, functions, packages, Java objects, libraries, types, indextypes, and user-defined operators.

INDEX

Grants the privilege to create indexes on a table.

{ ON COMMIT REFRESH | QUERY REWRITE }

Grants the privilege to create a refresh-on-commit materialized view or to create a materialized view for query rewrite on the specified table. It is available for materialized views only.

{ READ | WRITE }

Grants the privilege to read and write files against a named directory including full path name of the operating system directory. Since Oracle has the ability to store files outside of the database, the Oracle server process must run under a user context with privileges on the applicable directories. You can enable Oracle to enforce security on individual users via this security mechanism. Note that *WRITE* is only useful in connection with an external table, such as a logfile or an error file.

REFERENCES

Grants the privilege to define referential integrity constraints, and is available for use on tables.

{SELECT | INSERT | DELETE | UPDATE}

Grants the privilege to execute each of the respective SQL commands against the named schema object and is available for use on tables, views, sequences (*SELECT* only), and materialized views (*SELECT* only). Note that you must grant *SELECT* privileges to a user or role that requires *DELETE* privileges. You can grant column-level privileges, as a parentheses-encapsulated list following the object name, only when granting *INSERT*, *REFERENCES*, or *UPDATE* object privileges on a table or view.

UNDER

Grants the privilege to create a subview under the named view and is available for use on views and types only.

Grants the specified Oracle system privilege to one or more users or roles. For example, you could grant the *CREATE TRIGGER* privilege or the *ALTER USER* privilege. In each case, issuing a grant for a system privilege empowers the user or role to execute the command of the same name. Refer to [Table 3-2](#) later in this section for a full list of system privileges.

role

Grants a role to the user or to another role. In addition to user-defined roles, there are a number of system roles that come predefined with Oracle:

CONNECT, RESOURCE, and DBA

Provided for backward compatibility with older versions of Oracle. Do not use these roles in current or newer versions of Oracle since they may be dropped in the future.

DELETE_CATALOG_ROLE, EXECUTE_CATALOG_ROLE, and SELECT_CATALOG_ROLE

Grant members of these roles the privilege deleting, executing, and selecting from data dictionary views and packages.

EXP_FULL_DATABASE and IMP_FULL_DATABASE

Grant members of these roles the privilege of executing the import and export utilities.

AQ_USER_ROLE and AC_ADMINISTRATOR_ROLE

Grant members of these roles the privilege to use or administrate Oracle's Advanced Queuing functionality.

SNMPAGENT

Assigned only to Oracle Enterprise Manager and Intelligent Agent.

RECOVERY_CATALOG_OWNER

Grants the privilege to create users who own their own recovery catalog.

HS_ADMIN_ROLE

Grants the privilege to access areas of the data dictionary used to support Oracle's heterogeneous services feature

ON *schema_name*

Grants the named privilege to a user or role on an object in the schema. Database objects include tables, views, sequences, stored procedures, user-defined functions, packages, materialized views, user-defined types, libraries, indextypes, user-defined operators, or a synonym for any of these. If you do not include the schema name, Oracle assumes the schema of the current user.

Oracle also supports two additional keywords for special cases:

DIRECTORY

Grants privileges on a directory object, which is an Oracle object corresponding to a filesystem directory.

JAVA

Grants privileges on a Java *SOURCE* or a *RESOURCE* schema object.

TO { *grantee_name* | *role_name* | PUBLIC } [WITH { GRANT | HIERARCHY } OPTION]

Names the user or role who will get the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role. Multiple grantees can be listed with a comma between each, as in the following:

WITH GRANT OPTION

Enables the grantee to grant the specific privilege(s) to other users or to *PUBLIC*, but not to roles.

WITH HIERARCHY OPTION

Enables the grantee to receive privileges on all sub-objects when granted privileges on a parent object. This includes sub-objects created in the future. You can use this option only when granting the *SELECT* object privilege.

IDENTIFIED BY *password* [WITH ADMIN OPTION]

Establishes or changes a password that the grantee must use to enable a *role* being granted.

WITH ADMIN OPTION

Enables the grantee to exercise control over a role that you are granting to that grantee. First, this clause enables the grantee to grant and revoke the role to and from other users or non-*GLOBAL* roles. It further allows the grantee to drop the role, or to change the authorization needed to access a role.

Grants to users take effect right away. Grants to roles take effect right away, assuming the roles are enabled. Otherwise, a grant to a role takes effect as the role is enabled. Note that you can grant roles to users and to other roles (including *PUBLIC*). For example:

```
GRANT sales_reader TO sales_manager;
```

To grant privileges on a view, you must have the specific privilege and the *WITH GRANT OPTION* on all of the base tables for the view.

Any time you want a privilege available to all users, simply grant it to *PUBLIC*:

```
GRANT SELECT ON work_schedule TO public;
```

However, there are some restrictions when granting system privileges and roles:

- A privilege or role should not appear more than once in the *GRANT* statement.
- Roles cannot be granted to themselves.
- Roles cannot be granted recursively. That is, you couldn't grant the role **sales_reader** to **sales_manager** and then grant **sales_manager** to **sales_reader**.

You can grant multiple privileges of the same type in a single statement. However, the grants must be on the same type of objects:

```
GRANT UPDATE (emp_id, job_id), REFERENCES (emp_id),  
ON employees  
TO sales_manager;
```

As an aside, granting any of the object permissions on a table allows the user (or role) to lock the table in any lock mode using Oracle's *LOCK TABLE* statement.

Nearly every supported Oracle feature or command is assignable with a *GRANT* command (as shown in [Table 3-2](#)). Privileges can be granted not only on database objects (such as tables and views) and system commands (such as *CREATE ANY TABLE*), but also on schema objects (such as *DIRECTORY*, *JAVA SOURCE*, and *RESOURCE*).

The *ANY* option, shown in [Table 3-2](#), grants the privilege to execute a given statement against objects of a specific type

owned by any user within the schema. Without the *ANY* option, the user can only execute the statement against objects in their own schema. A more complete list of Oracle system privileges is shown in [Table 3-2](#).

Table 3-2. Oracle system privileges

Options	Description
<i>CREATE CLUSTER</i>	Grants privilege to create a cluster in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY CLUSTER</i>	Grants privilege to create, alter, or drop a cluster in any schema, respectively.
{ <i>CREATE DROP</i> } <i>ANY CONTEXT</i>	Grants privilege to create or drop any context namespace, respectively.
<i>ALTER DATABASE</i>	Grants privilege to alter the database.
<i>ALTER SYSTEM</i>	Grants privilege to issue <i>ALTER SYSTEM</i> statements.
<i>AUDIT SYSTEM</i>	Grants privilege to issue <i>AUDIT sql_statements</i> statements.
<i>CREATE DATABASE LINK</i>	Grants privilege to create private database links in grantee's own schema.
<i>CREATE PUBLIC DATABASE LINK</i>	Grants privilege to create public database links.
<i>DROP PUBLIC DATABASE LINK</i>	Grants privilege to drop public database links.
<i>DEBUG CONNECT SESSION</i>	Grants privilege to connect the current session to the Java Debug Wire Protocol (JDWP) debugger.
<i>DEBUG ANY PROCEDURE</i>	Grants the privilege to debug all PL/SQL objects in the database, similar to granting <i>DEBUG</i> on all procedures, functions, and packages.
<i>CREATE DIMENSION</i>	Grants privilege to create dimensions in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY DIMENSION</i>	Grants privilege to create, alter, or drop dimensions in any schema.
{ <i>CREATE DROP</i> } <i>ANY DIRECTORY</i>	Grants privilege to create or drop directory database objects.
<i>CREATE ROLE</i>	Grants the privilege to create a database role.
<i>GRANT ANY ROLE</i>	Grants the privilege to grant a role to other users.
<i>CREATE INDEXTYPE</i>	Grants privilege to create an indextype in the grantee's own schema.
{ <i>CREATE DROP EXECUTE</i> } <i>ANY INDEXTYPE</i>	Grants privilege to create, alter, drop, or execute an indextype in any schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY INDEX</i>	Grants privilege to create, alter, or drop an index in any schema or an index on any table in any schema.
{ <i>CREATE DROP</i> } <i>ANY LIBRARY</i>	Grants privilege to create or drop external procedure/function libraries in any schema.
<i>CREATE MATERIALIZED VIEW</i>	Grants privilege to create a materialized view in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> }	

<i>ANY MATERIALIZED VIEW</i>	Grants privilege to create, alter, or drop materialized views in any schema.
<i>GLOBAL QUERY REWRITE</i>	Enables rewrite using a materialized view, or creates a function-based index, when materialized view or index reference tables or views are in any schema.
<i>QUERY REWRITE</i>	Enables rewrite using a materialized view, or creates a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
<i>ON COMMIT REFRESH</i>	Creates a refresh-on-commit materialized view on any table in the database or alters any refresh-on-demand materialized view into a refresh-on-commit materialized view.
<i>FLASHBACK ANY TABLE</i>	Grants the privilege to issue a SQL flashback query on any table, view, or materialized view in any schema. This privilege is not required for <i>DBMS_FLASHBACK</i> procedures.
<i>CREATE OPERATOR</i>	Grants privilege to create an operator and its bindings in the grantee's own schema.
{ <i>CREATE DROP EXECUTE</i> } <i>ANY OPERATOR</i>	Grants privilege to create, drop, or execute an operator and its bindings in any schema.
{ <i>CREATE DROP ALTER</i> } <i>ANY OUTLINE</i>	Grants privilege to create, alter, or drop outlines that can be used in any schema that uses outlines.
<i>SELECT ANY OUTLINE</i>	Grants the privilege to clone private outlines from public ones.
<i>CREATE PROCEDURE</i>	Grants privilege to create stored procedures, functions, and packages in grantee's own schema.
{ <i>CREATE ALTER DROP EXECUTE</i> } <i>ANY PROCEDURE</i>	Grants privilege to create, alter, drop, or execute stored procedures, functions, and packages in any schema.
{ <i>CREATE ALTER DROP</i> } <i>PROFILE</i>	Grants privilege to create, alter, or drop profiles.
<i>ALTER RESOURCE COST</i>	Sets costs for session resources.
{ <i>ALTER DROP</i> } <i>ANY ROLE</i>	Grants privilege to create, alter, drop, or grant roles in the database.
{ <i>CREATE ALTER DROP</i> } <i>ROLLBACK SEGMENT</i>	Grants privilege to create, alter, or drop rollback segments.
<i>CREATE SEQUENCE</i>	Grants privilege to create sequences in grantee's own schema.
{ <i>CREATE ALTER DROP SELECT</i> } <i>ANY SEQUENCE</i>	Grants privilege to create, alter, drop, or select sequence in any schema in the database.
<i>CREATE SESSION</i>	Grants privilege to connect to the database.
<i>ALTER SESSION</i>	Grants privilege to issue <i>ALTER SESSION</i> statements.
<i>RESTRICTED SESSION</i>	Logs on after the instance is started using the SQL*Plus <i>STARTUP RESTRICT</i> statement.
<i>CREATE SNAPSHOT</i>	Grants privilege to create snapshots in grantee's own schema.
{ <i>CREATE ALTER DROP</i> } <i>ANY SNAPSHOT</i>	Grants privilege to create, alter, or drop snapshots (a.k.a. materialized views) in any schema.
<i>CREATE SYNONYM</i>	Grants privilege to create synonyms in grantee's own schema.

{CREATE DROP} ANY SYNONYM	Grants privilege to create or drop private synonyms in any schema.
{CREATE DROP} PUBLIC SYNONYM	Grants privilege to create or drop public synonyms.
CREATE ANY TABLE	Grants privilege to create tables in any schema. The owner of the schema containing the table must have a space quota on the tablespace to contain the table.
ALTER ANY TABLE	Grants privilege to alter any table or view in the schema.
BACKUP ANY TABLE	Uses the export utility to incrementally export objects from the schema of other users.
DELETE ANY TABLE	Deletes rows from tables, table partitions, or views in any schema.
DROP ANY TABLE	Grants privilege to drop or truncate tables or table partitions in any schema.
INSERT ANY TABLE	Grants privilege to insert rows into tables and views in any schema.
LOCK ANY TABLE	Grants privilege to lock tables and views in any schema.
UPDATE ANY TABLE	Grants privilege to update rows in tables and views in any schema.
SELECT ANY TABLE	Grants privilege to queries tables, views, or snapshots in any schema.
{CREATE ALTER DROP} TABLESPACE	Grants privilege to create, alter, or drop tablespaces.
MANAGE TABLESPACE	Grants privilege to take tablespaces offline and online, and begins and ends tablespace backups.
UNLIMITED TABLESPACE	Grants privilege to use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, the user's schema objects remain, but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
CREATE TRIGGER	Grants privilege to create a database trigger in grantee's own schema.
{CREATE DROP} ANY TRIGGER	Grants privilege to create or drop database triggers in any schema.
ALTER ANY TRIGGER	Grants privilege to enable, disable, or compile database triggers in any schema.
ADMINISTER DATABASE TRIGGER	Grants privilege to create a trigger on DATABASE. (You also must have the CREATE TRIGGER or CREATE ANY TRIGGER privilege.)
CREATE TYPE	Grants privilege to create object types and object-type bodies in grantee's own schema.
{CREATE ALTER DROP} ANY TYPE	Grants privilege to create, alter, or drop object types and object-type bodies in any schema.
EXECUTE ANY TYPE	Uses and references object types and collection types in any schema, and invokes methods of an object type in any schema <i>if you make the grant to a specific user</i> . If you grant EXECUTE ANY TYPE to a role, users holding the enabled role will <i>not</i> be able to invoke methods of an object type in any schema.
UNDER ANY TYPE	Creates subtypes under any parent object type, such as a table or view.
CREATE USER	Grants privilege to create users. This privilege also allows the creator to: <ul style="list-style-type: none"> • Assign quotas on any tablespace • Set default and temporary tablespaces • Assign a profile as part of a CREATE USER statement

<i>ALTER USER</i>	Grants the privilege to alter any user. This privilege authorizes the grantee to: <ul style="list-style-type: none"> • Change another user's password or authentication method • Assign quotas on <i>any</i> tablespace • Set default and temporary tablespaces • Assign a profile and default roles
<i>BECOME USER</i>	Grants privilege to become another user (required by any user performing a full database import).
<i>DROP USER</i>	Grants privilege to drop users.
<i>CREATE VIEW</i>	Grants privilege to create views in grantee's own schema.
<i>{CREATE DROP} ANY VIEW</i>	Grants privilege to create or drop views in any schema.
<i>UNDER ANY VIEW</i>	Creates subview under any parent view.
<i>ANALYZE ANY</i>	Analyzes any table, cluster, or index in any schema.
<i>AUDIT ANY</i>	Audits any object in any schema using <i>AUDIT schema_objects</i> statements.
<i>COMMENT ANY TABLE</i>	Allows comments on any table, view, or column in any schema.
<i>EXEMPT ACCESS POLICY</i>	Circumvents fine-grained access control. Generally, this is ill-advised since the user can then bypass any application-driven security policies.
<i>FORCE ANY TRANSACTION</i>	Forces the commit or rollback of any in-doubt distributed transaction in the local database and induces the failure of a distributed transaction.
<i>FORCE TRANSACTION</i>	Forces the commit or rollback of grantee's own in-doubt distributed transactions in the local database.
<i>GRANT ANY PRIVILEGE</i>	Grants any system privilege.
<i>GRANT ANY OBJECT PRIVILEGE</i>	Grants any object privilege available.
<i>RESUMABLE</i>	Enables resumable space allocations.
<i>SELECT ANY DICTIONARY</i>	Allows the user to query any data dictionary object in the SYS schema.
<i>SYSDBA</i>	Provides the user with <i>RESTRICTED SESSION</i> permission as well as authorizes the user to: <ul style="list-style-type: none"> • Perform <i>STARTUP</i> and <i>SHUTDOWN</i> operations • <i>ALTER DATABASE</i> • Open, mount, back up, or change character set • <i>CREATE DATABASE</i> • <i>ARCHIVELOG</i> and <i>RECOVERY</i>
<i>SYSOPER</i>	Provides the user with <i>RESTRICTED SESSION</i> permission as well as authorizes the user to: <ul style="list-style-type: none"> • Performs <i>STARTUP</i> and <i>SHUTDOWN</i> operations • <i>ALTER DATABASE, OPEN/MOUNT/BACKUP</i> • <i>ARCHIVELOG</i> and <i>RECOVERY</i>

Any of the privileges shown in [Table 3-2](#) containing the *ANY* keyword have special meaning. In effect, *ANY* gives a user the privilege to execute the specified command or operation in any schema. If you want to include all user schemas, but exclude the **SYS** system schema, keep the *O7_DICTIONARY_ACCESSIBILITY* initialization parameter at its default value of FALSE.

PostgreSQL

PostgreSQL supports a subset of the ANSI *GRANT* commands supporting object privileges against database objects such as tables. It does not support any of the *WITH* options available under the ANSI standard nor specialty features like *USAGE* and *UNDER*. PostgreSQL's *GRANT* syntax is as follows:

```
GRANT { ALL [PRIVILEGES]
| SELECT
| INSERT
| DELETE
| UPDATE
| RULE
| REFERENCES
| TRIGGERS
| USAGE } [,...]
ON [TABLE] { object_name } [,...]
TO {grantee_name | GROUP group_name | PUBLIC } [,...]
```

where:

ALL [PRIVILEGES]

Grants all privileges the grantor has available to grant. Using *ALL* is generally not a recommended approach, since it can encourage sloppy permissioning.

{SELECT | INSERT | DELETE | UPDATE}

Grants the specific privilege to a given user on the specified database object, such as a table or view. For column-level privileges, you may add a parenthetical list of the columns of a given table. The permission then applies only to the columns of the table that are named.

RULE

Grants the privilege to create or drop a rule on a table or view.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

TRIGGERS

Grants the privilege of creating a trigger to operate on a specific table and its columns.

USAGE

Grants privileges to use these objects: *CHARACTER SET*, *COLLATION*, *TRANSLATION*, and *DOMAIN* (user-defined datatype).

ON [TABLE] *object_name*

Specifies the object to which permissions are granted. The optional *TABLE* keyword is noise. You may grant permissions on more than one object at a time by specifying a comma-delimited list of object names.

TO {*grantee_name* | GROUP *group_name* | PUBLIC } [,...]

Names the user or role, or multiples thereof, that receive the specified privileges. In PostgreSQL, PUBLIC is a synonym for all users. PostgreSQL allows permissions to be assigned to a *GROUP* (essentially a role), provided it is a valid, pre-existing *group_name*.

Multiple privileges may be granted at one time, as long as commas separate them. Do not combine *ALL [PRIVILEGES]* with other privileges, but the remainder may be combined in any order.

PostgreSQL does not support the *WITH* clauses or column-level permissions. Currently, if you want column-level permissions, you must set those up using a view showing only the columns of a table that you want a user to access.

Under PostgreSQL, the creator of an object has all privileges on the object, though they may revoke their own privileges on an object. However, the privilege for the creator of an object to assign privileges on that object cannot be revoked. Similarly, the right to drop an object is permanently assigned to the creator of the object, and cannot be revoked. Users other than the creator of an object do not have permissions on an object until specifically granted.

PostgreSQL's implementation of *GRANT* behaves as if *WITH GRANT OPTION* is always enabled. Any user granted privilege to a table is able to grant their privileges to other users.

PostgreSQL's support for the *GRANT* statement is elementary. In the following examples, *INSERT* privileges are granted on the **publishers** table to the PUBLIC role and *SELECT* and *UPDATE* privileges are granted on the **sales** table to the users **Emily** and **Dylan**:

```
GRANT INSERT ON publishers TO PUBLIC;
```

```
GRANT SELECT, UPDATE ON sales TO emily, dylan;
```

SQL Server

SQL Server's implementation of *GRANT* offers few variations from the ANSI standard. It does not support the ANSI standard's *FROM* clause, nor the *HIERARCHY* or *ADMIN* options. But it does offer the ability to grant specific system privileges and to grant them under another user's security context. The syntax is as follows:

```
GRANT { [object_privilege] [,...] |
```

```
  [system_privilege] }
```

```
[ON { [object] [(column [,...])]
```

```
TO {grantee_name [,...] | role [,...] | PUBLIC | GUEST }
```

```
[WITH GRANT OPTION]
```

```
[AS {group | role}]
```

where:

object_privileges

Grants privileges for a variety of operations, which may all be combined (with the exception of the *ALL [PRIVILEGES]* privilege) in any order, as follows:

ALL [PRIVILEGES]

Grants all privileges currently assigned to the named users and/or for the specified database objects. This is generally not a recommended approach since it can encourage sloppy programming. *ALL* can only be used by members of the *SYSADMIN* and *DB_OWNER* system roles, or by the object owner.

{*SELECT* | *INSERT* | *DELETE* | *UPDATE*}

Grants the specific privilege to a given user on the specified database object, such as a table or view. For column-level privileges, add a parenthetical list of columns of the table. The privileges will then be assigned only for the named columns.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

EXECUTE

Grants the privilege to execute a stored procedure, user-defined function, or extended stored procedure.

system_privileges

Grants one of the following privileges, each giving the ability to execute the corresponding statement:

CREATE DATABASE
CREATE DEFAULT
CREATE FUNCTION
CREATE PROCEDURE
CREATE RULE
CREATE TABLE
CREATE VIEW
BACKUP DATABASE
BACKUP LOG

The privilege to issue a *CREATE* statement also implies the privilege to issue the equivalent *ALTER* or *DROP* statement.

ON { [*object*] [(*column* [,...])] }

Specifies the object for which a privilege is granted. This clause is not required when granting system privileges. If the object is a table or view, you may optionally grant privileges on specific columns. You may grant *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *REFERENCES* privileges on a table or view. You may grant only *SELECT* and *UPDATE* privileges on the columns of a table, view, or a user-defined function. You may grant *EXECUTE* privileges on a stored procedure or extended stored procedure. You may also grant *REFERENCES* privileges on user-defined functions. *REFERENCES* privileges are also required on functions or views created using the *WITH SCHEMABINDING* clause.

TO { *grantee_name* [,...] | *role* [,...] | PUBLIC | GUEST }

Names the user or role who will gain the privilege being granted. Multiple grantees can be listed in a comma-separated list. Use the keyword *PUBLIC* to grant privileges to the *PUBLIC* role, a role that implicitly includes all users. SQL Server also supports the role known as *GUEST*, a role used by all users who do not have any other user account in the database. Because SQL Server allows two different security models (one based on the database and one based on the Windows operating system), you may be granting privileges to a SQL Server user, a Windows user, a Windows group, or a SQL Server database role. The grantee name may be any of these.

WITH GRANT OPTION

Enables the grantee to further grant the given permission to other users or roles. You may use *WITH GRANT OPTION* only when granting object permissions.

AS {*group* | *role*}

Specifies an alternative user or group that has the authority in the current database to execute the *GRANT* command. You can use the *AS* clause to grant privileges as if the granting session is part of a different group or role context from the current one.

Privileges may not be granted in any database other than the current database, nor may privileges be granted in more than one database at a time.



SQL Server has a somewhat different security model than the other platforms described here (and somewhat different from the ANSI standard). As with the ANSI standard, SQL Server uses the *GRANT* command to assign specific privileges to a user or role, and the *REVOKE* command to remove those permissions. SQL Server augments these commands with the *DENY* command, using essentially the same syntax as *REVOKE*.

with the *DENY* command, using essentially the same syntax as *REVOKE*.

DENY, under SQL Server, allows DBAs to declare certain privileges as strictly off-limits to a user or role. Any privileges that have been denied must be revoked before they can be granted. As such, *DENY* takes precedence over *GRANT* and *REVOKE*. *DENY* can be used to effectively take away any permission from an individual that might otherwise inherit permissions from membership in any Windows groups or SQL Server database roles.

SQL Server applies precedence to permissions. Thus, if a user has a privilege granted at the user level, but the permission is revoked at the group level (when the user is a member of a group), the permission is revoked at both levels.

SQL Server also contains a number of fixed system roles with preset privileges on both objects and commands. The SQL Server system roles are:

SYSADMIN

Can perform any activity on the server and has access to all objects.

SERVERADMIN

Can set server configuration options as well as shut down and restart the server.

SETUPADMIN

Can start up procedures and linked servers.

SECURITYADMIN

Can read error logs, change passwords, administrate logins, and grant *CREATE DATABASE* permissions.

PROCESSADMIN

Can administrate processes running on the SQL Server.

DBCREATOR

Can create, alter, and drop databases.

DISKADMIN

Can administrate disks and files.

BULKADMIN

Can perform BCP operations and execute the *BULK INSERT* statement (and regular *INSERT* statement) against tables in the database.

The *SYSADMIN* system role can grant any permission in any database on the server. Members of the *DB_OWNER* and *DB_SECURITYADMIN* system roles can grant any permission on any statement or object in databases they own. Members of the *DB_DDLADMIN* system role, the *SYSADMIN* system role, and the database owner may grant system privileges.

Members of the system roles may add other logins or users to their roles, but not via a *GRANT* statement. Instead, to add a user to a system role, you must use a SQL Server system stored procedure called **sp_addsrvrolemember**.

In addition to the server-wide system roles, there are a number of database-wide system roles. That is, the previous system roles have privileges that span every database on the server, whereas the following system roles exist in every database and have privileges within the scope of a specific database, as follows:

DB_OWNER

Can perform all the functions of the other database system roles, plus perform other maintenance and configuration tasks for the database.

DB_ACCESSADMIN

Can add or remove Windows and SQL Server users and groups to the database.

DB_DATAREADER

Can read data from all user tables in the database.

DB_DATAWRITER

Can read, add, change, or delete data from all user tables in the database.

DB_DDLADMIN

Can add, change, and drop objects in the database, as well as run all DDL statements.

DB_SECURITYADMIN

Can administrate SQL Server roles and users as well as statement and object permissions throughout the database.

DB_BACKUPOPERATOR

Can perform backups of the database.

DB_DENYDATAREADER

Denies permission to read data in the database.

DB_DENYDATAWRITER

Denies permission to change data in the database.

Like the server-wide system roles, database-wide system roles cannot be granted to a user via the *GRANT* statement. Instead, you must use a SQL Server system stored procedure called **sp_addrolemember**.

In the following example, the *CREATE DATABASE* and *CREATE TABLE* system privileges are granted to the users **emily** and **sarah**. Next, numerous permissions are granted on the **titles** table to the editors group. The editors are then able to grant permission to others:

```
GRANT CREATE DATABASE, CREATE TABLE TO emily, sarah
```

```
GO
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON titles
```

```
TO editors
```

```
WITH GRANT OPTION
```

```
GO
```

The following example grants permissions to the database user **sam** and the Windows user **jacob**:

```
GRANT CREATE TABLE TO sam, [corporate\jacob]
```

```
GO
```

The following example shows how to grant permissions using the optional *AS* keyword. In this example, the user **emily**

owns the **sales_detail** table and she grants *SELECT* privileges to the **sales_manager** role. The user **kelly**, who is a member of the **sales_manager** role, wants to grant *SELECT* privileges to **sam**, but cannot because the permissions were granted to the **sales_manager** role and not to her explicitly. **kelly** can use the *AS* clause to get around this hurdle:

```
-- Initial grant
```

```
GRANT SELECT ON sales_detail TO sales_manager
```

```
WITH GRANT OPTION
```

```
GO
```

```
-- Kelly passes the privilege to Sam as a member of the
```

```
    sales_manager role
```

```
GRANT SELECT ON sales_detail TO sam AS sales_manager
```

```
GO
```

See Also

REVOKE

IN Operator

The *IN* operator provides a way to delineate a list of values, either explicitly listed or from a subquery, and compare a value against that list in a *WHERE* or *HAVING* clause. In other words, it gives you a way to say "Is value A *in* this list of values?"

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
{WHERE | HAVING | {AND | OR} }
```

```
    value [NOT] IN ({comp_value1, comp_value2 [...] | subquery } )
```

Keywords

```
{WHERE | HAVING | {AND | OR} } value
```

Permitted under either the *WHERE* or *HAVING* clauses. The *IN* comparison may also be a part of a *AND* or *OR* clause in a multicondition *WHERE* or *HAVING* clause. *value* may be of any datatype, but is usually the name of a column of the table referenced by the transaction, or perhaps a host variable when used programmatically.

NOT

Optionally tells the database to look for a result set that contains values that are not in the list.

IN ({ *comp_value1*, *comp_value2* [,...] | *subquery* })

Defines the list of comparative values (hence, *comp_value*) to compare against. Each *comp_value* must be of the same or compatible datatype as the initial *value*. They are also governed by standard datatype rules. For example, string values must be delimited by quotes, while integer values need none. As an alternative to listing specific values, you may use parentheses to enclose a subquery that returns one or more values of a compatible datatype.

In the following example, generated on SQL Server, we look for all employees in the **employee** table of the **HR** database who have a home state of Georgia, Tennessee, Alabama, or Kentucky:

```
SELECT *  
FROM hr..employee  
WHERE home_state IN ('AL','GA','TN','KY')
```

Similarly, we can look for all **employees** in the **HR** database who are authors in the **PUBS** database:

```
SELECT *  
FROM hr..employee  
WHERE emp_id IN (SELECT au_id FROM pubs..authors)
```

You can also use the *NOT* keyword to return a result set based upon the absence of a value. In the following case, the company headquarters is located in New York, and many workers commute in from neighboring states. We want to see all such workers:

```
SELECT *  
FROM hr..employee  
WHERE home_state  
NOT IN ('NY','NJ','MA','CT','RI','DE','NH')
```



Note that Oracle, while fully supporting the ANSI functionality, extends the functionality of the *IN* operator by allowing multiple argument matches. For example, the following *SELECT...WHERE ...IN* statement is acceptable on Oracle:

```
SELECT *  
FROM hr..employee e  
WHERE (e.emp_id, e.emp_dept) IN  
( (242, 'sales'), (442, 'mfg'), (747, 'mkt') )
```

See Also

ALL/ANY/SOME
BETWEEN
EXISTS
LIKE
SELECT
SOME/ANY

INSERT Statement

The *INSERT* statement adds rows of data to a table or view.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported, with variations

The *INSERT* statement allows rows to be written to a table through one of several methods:

- The first method is to insert one or more rows using the *DEFAULT* values specified for a column via the *CREATE TABLE* or *ALTER TABLE* statements.
- The second and most common method is to declare the actual values to be inserted into each column of the record.
- The third method, which populates a table with many records at a time, is to insert the result set of a *SELECT* statement.

SQL2003 Syntax

```
INSERT INTO [ONLY] {table_name | view_name} [(column1 [,...])]  
[OVERRIDE {SYSTEM | USER} VALUES]  
{DEFAULT VALUES | VALUES (value1 [,...]) | SELECT_statement}
```

Keywords

[ONLY]

Used on typed tables only, this optional keyword ensures that the values inserted into *table_name* do not insert into any subtables.

{*table_name* | *view_name*} [(*column1* [,...])]

Declares the updateable target table or view into which the records will be inserted. You must have *INSERT* privileges on the table or, at a minimum, on the columns that will receive the inserted values. If no schema information is included, as in **scott.employee**, then the current schema and user context are assumed. You may optionally include a list of the columns in the target table or view that will receive data.

OVERRIDE {SYSTEM | USER} VALUES

Requires the *SYSTEM* keyword when inserting a literal *value* into a column that would otherwise be given a system-generated value, such as autogenerated sequence number. The *OVERRIDE USER VALUES* clause does the converse, by inserting system-supplied values even if a user has provided literal values to insert.

DEFAULT VALUES

Inserts all values declared via the *DEFAULT* column characteristic on the target table where they exist, and

NULLs where they do not exist. (Of course, the *DEFAULT* column characteristic can specify *NULL* as a value too.) This operation inserts a single record. It might encounter an error depending on how the *PRIMARY KEY* constraint or *UNIQUE* constraint is constructed on the target table, assuming such constraints exist.

VALUES (*value1* [,...])

Specifies the actual values to be inserted into the target table. The number of values must match the exact number of columns in the column list. Furthermore, the values must be datatype- and size-compatible with the columns of the target table. Each value in the value list corresponds to the column with the same ordinal number in the column list. Thus, the first column will get its data from the first value, the second column from the second value, and so on until all columns are satisfied. You may also optionally use the keywords *DEFAULT* to insert a column's default value and *NULL* to insert a null value.

SELECT_statement

Inserts rows retrieved via the *SELECT* statement into the target table or target view. The values retrieved by the fully formed *SELECT* statement's select item list correspond directly to the columns of the column list. The target table or view may not be referenced in the *SELECT* statement's *FROM* or *JOIN* clauses.

Rules at a Glance

You may insert values into tables and into views built upon a single source table. The *INSERT . . . VALUES* statement adds a single row of data to a table using literal values supplied in the statement. In the following example, a new row in the **authors** table is inserted for the author **Jessica Rabbit**:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone,  
    address, city, state, zip, contract )  
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,  
    '1717 Main St', NULL, 'CA', '90675', 1)
```

Every column in the table is assigned a specific, literal value except the **phone** column, which is assigned the default value (as defined during the *CREATE TABLE* or *ALTER TABLE* statement), and the **city** column, which is set to *NULL*.

An important concept to remember is that you may skip columns in the table and set them to *NULL*, assuming they allow *NULL* values. Inserts that leave some columns *NULL* are called partial *INSERTs*. Here is a partial *INSERT* that performs the same work as the preceding example:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone, contract )  
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT, 1)
```

The *INSERT* statement, combined with a nested *SELECT* statement, allows a table to be quickly populated with one or more rows from the result set of the *SELECT* statement. When using *INSERT . . . SELECT* between a target table and a source table, it is important to ensure that the datatypes returned by the *SELECT* statement are compatible with the datatypes in the target table. For example, to load data from the **sales** table into the **new_sales** table:

```
INSERT INTO sales (stor_id, ord_num, ord_date, qty, payterms,  
    title_id)  
SELECT  
    CAST(store_nbr AS CHAR(4)),  
    CAST(order_nbr AS VARCHAR(20)),  
    order_date,  
    quantity,  
    SUBSTRING(payment_terms,1,12),  
    CAST(title_nbr AS CHAR(1))  
FROM new_sales
```

```
WHERE order_date >= 01-JAN-2005
```

```
-- retrieve only the newer records
```

You must specify the columns in the table that receive data by enclosing their names in parentheses, in a comma-delimited list. This **column_list** can be omitted, but all columns that are defined for the table are then assumed, in their ordinal position. Any column with an omitted value will be assigned its default value (according to any *DEFAULT* column setting on the table) or, if no *DEFAULT* column setting exists, NULL. The columns in the column list may be in any order, but you may not repeat any column in the list. Furthermore, the columns and their corresponding *value* entries must agree in terms of datatype and size.

The first of the following two examples leaves off the column list, while the second uses only *DEFAULT* values:

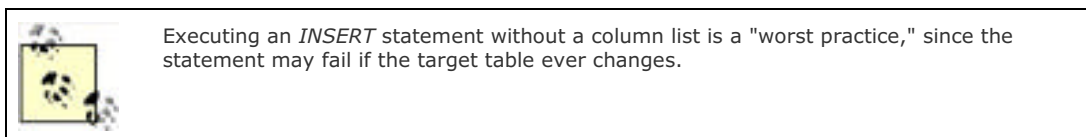
```
INSERT INTO authors
```

```
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,  
       '1717 Main St', NULL, 'CA', '90675', 1)
```

```
INSERT INTO temp_details
```

```
DEFAULT VALUES
```

The first statement would succeed only as long as all the values in the value list correspond correctly with the datatypes and size limitations of the columns of the target table. Any inconsistencies would generate an error. The second statement would succeed only as long as defaults were declared for the columns of the target table, or those columns allowed NULL values.



Programming Tips and Gotchas

INSERT statements will *always* fail under the following circumstances:

- When a datatype mismatch occurs between a column and its value
- When a column is defined as not NULL and the insertion value is NULL
- When a duplicate value is inserted into a *UNIQUE* or *PRIMARY KEY* constraint
- When the inserted values do not meet the requirements of a *CHECK* constraint
- When an inserted value is constrained by a *FOREIGN KEY* constraint because the value is not derived from the declared primary key of another table

The most common error encountered when executing *INSERT* statements is a mismatch between the number of columns and the number of values. Accidentally leave out a value that corresponds to a column, and you are likely to encounter an error that will cause the statement to fail.

INSERT statements also fail when an inserted value is of a datatype that is a mismatch with the column of the target table. For example, an attempt to insert a string like "Hello world" into an integer column would fail. On the other hand, some database platforms automatically and implicitly convert certain datatypes. For example, SQL Server will automatically convert a date value to a character string for insertion into a *VARCHAR* column.

Another common problem encountered with the *INSERT* statement is a size mismatch between a value and its target column. For example, inserting a long string into a *CHAR(5)* target column, or inserting a very large integer into a *TINYINT* column, can cause problems. Depending on the platform you are using, the size mismatch may cause an outright error and rollback of the transaction or the database server may simply trim the extra data. Either result is undesirable. Similarly, a problem can arise when an *INSERT* statement attempts to insert a NULL value into a target column that does not accept NULL.



Most problems with *INSERT* statements occur because the programmer does not know the target table very well. Make sure you understand the target table or view before writing elaborate *INSERT* statements.

DB2

DB2 supports the ANSI standard form of *INSERT*, with the exception of the *OVERRIDE* clause, and includes some added syntax to control transaction isolation settings:

```
INSERT INTO table_name [(column1 [...])]
{VALUES | VALUES (value1 [...]) | SELECT_statement }
[ WITH { RR | RS | CS | UR }
```

where:

```
INSERT INTO table_name [(column1 [...]) ]
```

Identifies the target table, view, or nickname on the DB2 server. The target table or view may be a standard or typed table or view. The view may be based on a *SELECT* statement containing a *UNION ALL* clause, as long as the inserted row matches the check constraint(s) of exactly one table.

WITH

Controls transaction isolation level for the update statement. *RR* is repeatable read, *RS* is read stability, *CS* is cursor stability, and *UR* is uncommitted read. See *SET TRANSACTION ISOLATION LEVEL* for full details on ANSI transaction isolation levels.

DB2 strictly enforces these rules:

- Values inserted must be compatible with the column datatype
- Integer and string values must not exceed the storage capability of the target column
- Inserted values must pass any check or unique constraints
- Values inserted into a foreign key column must exist in a corresponding primary key elsewhere in the database

DB2 supports identity columns; that is, columns that automatically generate a monotonically increasing value. When inserting into an identity column, DB2 generates a new value for the identity column. Identity columns are defined as either *GENERATED ALWAYS* or *GENERATED BY DEFAULT* upon creation. If defined as *GENERATED ALWAYS*, you should not specify a value for the column — leave it to default. If defined as *GENERATED BY DEFAULT*, you may insert your own value (explicitly or via a subquery), or you can allow DB2 to generate the value for you. In the following examples, assume the **emp_id** column is an identity column with the *GENERATED ALWAYS* property:

```
INSERT INTO employee (emp_id, emp_lname, emp_fname)
```

```
VALUES (DEFAULT, :var_lname, :var_fname)
```

```
INSERT INTO employee (emp_lname, emp_fname)
```

```
VALUES (:var_lname, :var_fname)
```

Both of the example statements shown above will work without error.

MySQL

MySQL supports several *INSERT* syntax options that engender MySQL's reputation for high speed:

```
INSERT [LOW_PRIORITY | DELAYED] [IGNORE]
[INTO] [[database_name.]owner.] table_name [(column1 [,...])]
{VALUES (value1 [,...]) | SELECT_statement |
SET column1=value1, column2=value2 [,...]}
```

where:

LOW_PRIORITY

Defers the execution of *INSERT* until no other clients are reading from the table. This could result in a long wait. *LOW_PRIORITY* should not be used with MyISAM tables because it disables concurrent inserts.

DELAYED

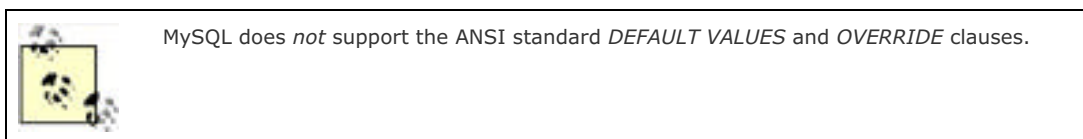
Allows the client to continue at once, even if the *INSERT* has not yet completed.

IGNORE

Tells MySQL not to attempt to insert records that would duplicate a value in a primary key or unique key; otherwise, without this clause, the *INSERT* fails when such duplication occurs. If a duplicate is encountered while the *IGNORE* clause is in use, the duplicate records are ignored while all the other records are inserted.

SET *column=value*

An alternate syntax that allows you to specify values for target columns by name.



MySQL trims any portion of a value that has a size or datatype mismatch. Thus, inserting "10.23 X" into a decimal datatype column will cause "...X" to be trimmed from the inserted value. If you attempt to insert a numeric value into a column that is beyond the range of the column, MySQL will trim the value. Inserting an illegal time or date value in a column will result in a zero value for the target column. Inserting the string "Hello World" into a *CHAR(5)* column will result in the value being trimmed to just the five characters in "Hello". This string trimming feature applies to *CHAR*, *VARCHAR*, *TEXT*, and *BLOB* columns.

Oracle

Oracle's implementation of the *INSERT* statement allows data insertion into a given table, view, partition, subpartition, or object table. It also supports additional extensions such as inserting records into many tables at once and conditional inserts. The syntax is:

-- standard *INSERT* statement

```
INSERT [INTO] {table_name [ [SUB]PARTITION (prt_n_name)] |
(subquery) [WITH {READ ONLY | CHECK OPTION
[CONSTRAINT constr_name ]}] |
TABLE (collection) [ (+) ] } [alias]
[(column1 [,...])]
{VALUES (value1 [,...]) [RETURNING expression1 [,...]]
INTO variable1 [,...]} |
```

```
SELECT_statement [WITH {READ ONLY |  
CHECK OPTION [CONSTRAINT constr_name]} }  
-- conditional INSERT statement  
INSERT {[ALL | FIRST]} WHEN condition  
THEN standard_insert_statement  
ELSE standard_insert_statement
```

where:

INSERT [INTO]

Inserts one or more rows into a single table, view, materialized view, or subquery. The *INTO* keyword is optional. You insert a single row using the *VALUES* clause and many rows using a *subquery*.

table_name [[SUB]PARTITION (*prtn_name*)]

Identifies the target into which you will insert data. The target may be a table, view, materialized view, or a subquery. To enable you to fully qualify the target, *table_name* can expand to [*schema*].*table_name*[*@db_link*].

You may optionally identify the schema and remote address (via *@db_link*) of the target, but the current schema and local database are assumed if you do not otherwise specify. You may optionally identify the *PARTITION* or *SUBPARTITION*, through the *prtn_name* parameter, into which the record(s) will be inserted, as long as the target is not an object table or object view.

subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT *constr_name*]}]

Instructs Oracle to insert records into the base table or tables of the *subquery*, where the subquery is a normally formed *SELECT* statement. Essentially, you're using a subquery to construct a view on-the-fly, and the effect is the same as inserting into a view. This is your primary means for inserting values into multiple tables at one time. All of the columns defined by the subquery, across all tables, must have a corresponding value to insert or else a failure will occur. Multitable inserts must use the subquery format. The following options apply to when using subqueries:

WITH READ ONLY

Indicates the subqueried table or view cannot be updated until the statement completes.

*WITH CHECK OPTION [CONSTRAINT *constr_name*]*

Indicates that you cannot insert rows into the table or view that would not pass the *constr_name* check constraint.

TABLE (collection) [(+)] } [alias]

Tells Oracle that the *collection* should be treated like a standard target (i.e., a table or view), whether it be a subquery, a column, a function, or a collection constructor. In any event, the table collection must return a nested table or *VARRAY* set of values. Since constructions can be very long, you can provide an optional *alias*. Aliases are not allowed in multitable insert operations.

[(column1 [,...])]

Specifies the target columns into which data will be inserted. If you leave off the columns, Oracle assumes that the *VALUES* clause or columns of the subquery will perfectly match the columns of the target. Oracle will return an error if you do not insert a value for any columns marked as NOT NULL that do not have a defined, default value.

*VALUES (value1[,...]) [RETURNING *expression1* [,...]] INTO *variable1* [,...]*

Inserts values into the target table or tables. As with the ANSI standard, there must be a matching value for every column, though the value can be *DEFAULT*, or, if the column accepts nulls, the value can be the literal NULL. *DEFAULT* is not allowed when inserting into a view. On multitable insert operations, the *VALUES* clause must return a corresponding value for every item in the select list of the subquery. The syntax is as follows:

RETURNING *expression1*

Retrieves the rows inserted by the operation. The *expression* returned by the statement is often a value being inserted, but could be another value. For example, you might use the returning clause to find the value of an automatically generated primary key. Single row operations store the results into host variables or PL/SQL variables, while multirow operations store them in bind arrays. You can use *RETURNING* against tables, views with a single base table, and materialized views. The *RETURNING* clause is not allowed with multitable insert operations.

INTO *variable1*

Specifies the variables that will hold the values returned as a result of the *RETURNING* clause. You must declare a corresponding host variable or PL/SQL variable for each expression in the *RETURNING* clause. You cannot use the *INTO* clause to hold a *LONG* datatype, with remote objects, on views that have *INSTEAD OF* triggers, or with parallel *INSERT*, *UPDATE*, or *DELETE* statements.

ALL

Performs a multitable *INSERT* statement. *ALL* is used only with the subquery format. Without a *WHEN* clause, *ALL* unconditionally inserts all the data retrieved by the subquery into the tables defined. *ALL*, with a *WHEN* clause, performs conditional insert operations that tell Oracle to evaluate all *WHEN* clauses regardless of the results of any other *WHEN* operation. Each time a *WHEN* clause evaluates as TRUE, Oracle executes the corresponding *INTO* clause. Multitable inserts are not parallelized on index-organized tables or bitmap indexed tables. They are not allowed at all when:

- The target is a view or materialized view
- The target is a remote table
- The *INSERT* command uses a table collection expression
- The table needs more than 999 total target columns
- The subquery uses a sequence

FIRST

Tells Oracle to evaluate the *WHEN* clauses in order and, when it finds the first TRUE expression, execute the corresponding *INTO* clause and skip all other *WHEN* clauses.

WHEN *condition* THEN *standard_insert_statement*

Sets a *condition* and, when the condition is TRUE, executes the *THEN* insert clause. The value of *condition* is evaluated for each column returned in the result set of the subquery. Up to 127 *WHEN* clauses are allowed.

ELSE *standard_insert_statement*

Executes when no *WHEN* clause evaluates as TRUE.

Oracle allows the standard *INSERT* operations as described under the ANSI implementation, such as *INSERT...SELECT* and *INSERT...VALUES*. But it has a great many special variations.

When inserting into tables that have assigned sequences, but sure to use the **sequence_name.nextval** function call to insert the next logical number in the sequence. For example, assume you want to use the **authors_seq** sequence to set the value of **au_id** when inserting a new row into the **authors** table:

```
INSERT authors (au_id, au_lname, au_fname, contract )
```

```
VALUES (authors_seq.nextval, 'Rabbit', 'Jessica', 1)
```


When retrieving values during an *INSERT* operation, check for a one-for-one match between the expressions in the *RETURNING* clause and the variables of the *INTO* clause. The expressions returned by the clause do not necessarily have to be those mentioned in the *VALUES* clause. For example, the following *INSERT* statement places a record into the **sales** table, but places a completely distinct value into a bind variable:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1)
RETURNING hire_date INTO :temp_hr_dt;
```

Notice that the *RETURNING* clause returns the **hire_date** even though **hire_date** is not one of the values listed in the *VALUES* clause. (In this example, it is reasonable to assume a default value was established for the **hire_date** column.)

An unconditional multitable *INSERT* statement into a lookup table that contains a list of all the approved **jobs** in our company looks like this:

```
INSERT ALL
  INTO jobs(job_id, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+1, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+2, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+3, job_desc, min_lvl, max_lvl)
  VALUES(job_id, job_desc, min_lvl, max_lvl)
SELECT job_identifier, job_title, base_pay, max_pay
FROM   job_descriptions
WHERE  job_status = 'Active';
```

And just to make things more complex, Oracle allows multitable *INSERT* statements that are conditional:

```
INSERT ALL
  WHEN job_status = 'Active'   INTO jobs
  WHEN job_status = 'Inactive' INTO jobs_old
  WHEN job_status = 'Terminated' INTO jobs_cancelled
  ELSE INTO jobs
SELECT job_identifier, job_title, base_pay, max_pay
FROM   job_descriptions;
```

Note that in the above example, you would have to follow each *INTO* clause with a *VALUES* clause if you were skipping *NOT NULL* columns in the target table. The following example shows this syntax:

```
INSERT FIRST
  WHEN job_status = 'Active'
  INTO jobs
  VALUES(job_id, job_desc, min_lvl, max_lvl)
```

```
WHEN job_status = 'Inactive'  
  
    INTO jobs_old  
  
    VALUES(job_id, job_desc, min_lvl, max_lvl)  
  
WHEN job_status = 'Terminated'  
  
    INTO jobs_cancelled  
  
    VALUES(job_id, job_desc, min_lvl, max_lvl)  
  
WHEN job_status = 'Terminated'  
  
    INTO jobs_outsourced  
  
    VALUES(job_id, job_desc, min_lvl, max_lvl)  
  
ELSE INTO jobs  
  
    VALUES(job_id, job_desc, min_lvl, max_lvl)  
  
SELECT job_identifier, job_title, base_pay, max_pay  
  
FROM job_descriptions;
```

Notice that in the above example, the *FIRST* clause also tells Oracle to execute the first occurrence of `job_status = 'Terminated'` by inserting the records into the **jobs_cancelled** table and skipping the **jobs_outsourced** *INSERT* operation.

Oracle allows you to insert data into a table, partition, or view (also known as the target) using a regular or *direct-path INSERT* statement. Under a regular insert, Oracle maintains referential integrity and reuses free space in the target. Under direct-path, Oracle appends data at the end of the target table without filling in any free space gaps elsewhere in the table. This method bypasses the buffer cache and writes directly to the datafiles; hence the term "direct-path."



Oracle allows the use of hints to circumvent default query optimization for *INSERT* statements. For example, you can use the *APPEND* hint to ensure that an *INSERT* uses a direct-path insert. Refer to the platform documentation for more details on hints that are usable with *INSERT*.

Direct-path enhances performance on long, multirecord insert operations. However, the following is a list of restrictions that cause an *INSERT* statement to perform a regular *INSERT*, instead of direct-path:

- Altering the data in the target with an *UPDATE* or *DELETE* statement before the *INSERT* statement in a single transaction. (*UPDATE* and *DELETE* are allowed after the direct-path *INSERT* statement.)
- The *INSERT* statement is or may become distributed.
- The target contains a *LOB* or object datatype column.
- The target has a clustered index or index-organized table.
- The target has triggers or referential integrity constraints.
- The target is replicated.
- The *ROW_LOCKING* initialization parameter is set to *INTENT*.

In addition, Oracle will not allow you to query (e.g., using *SELECT*) a table later in the same transaction after you perform a direct-path *INSERT* into that table until a *COMMIT* has been performed.

Oracle allows you to use parallel direct-path inserts into multiple tables, but you may only use subqueries to insert the data into the tables, not a standard *VALUES* clause.



Inserting *LOBs* and *BFILES* is tricky. You should initialize such values to *NULL* before inserting. *RAW* columns are also tricky. If you insert a regular string into a *RAW* column, all future queries against the column will be forced to use a table scan.

PostgreSQL

PostgreSQL supports the ANSI standard for the *INSERT* statement. It does not support the ANSI SQL clause *OVERRIDE SYSTEM GENERATED VALUES*, but is otherwise identical in syntax and behavior. Refer to [SQL2003 Syntax](#), earlier, for the syntax and usage.



PostgreSQL attempts to perform automatic datatype coercion when the expressions in the *VALUE* clause or the select item list of a subquery do not match the datatypes defined for the target table or view.

SQL Server

SQL Server supports a few extensions to the ANSI standard for *INSERT*. It supports several rowset functions (explained below), as well as the capability to insert the results from stored procedures and extended procedures directly into the target table. SQL Server's syntax is:

```
INSERT [INTO] table_name [(column1 [,...])  
{[DEFAULT] VALUES |  
VALUES (variable1 [,...]) |  
SELECT statement |  
EXEC[UTE] proc_name { [[@param =] value {[OUTPUT]} [,... ] ] }
```

where:

INSERT INTO table_name

Tells SQL Server that the target is either a table, a view, or a rowset function. When inserting into a view, an *INSERT* cannot affect more than one of the base tables in the view, if there are more than one. Rowset functions allow SQL Server to source data from special or external data sources such as XML streams, full-text search file structures (a special structure in SQL Server used to store things like MS Word documents and MS PowerPoint slideshows within the database), or external data sources (like an MS-Excel spreadsheet). Examples are shown later in this section. SQL Server currently supports the following *rowset_functions* for the *INSERT* statement:

OPENQUERY

Executes a pass-through *INSERT* against a linked server. This is an effective means of performing a nested *INSERT* against a data source that is external to SQL Server. The data source must first be declared as a linked server.

OPENROWSET

Executes a pass-through *INSERT* statement against an external data source. This is similar to *OPENDATASOURCE*, except that *OPENDATASOURCE* only opens the data source, it does not actually pass through an *INSERT* statement. *OPENROWSET* is intended for occasional, ad hoc usage only.

column1 [,...]

Identifies one or more columns in the target table. The list must be enclosed in parentheses, and commas must separate each item in the list. SQL Server automatically provides values for *IDENTITY* columns, *TIMESTAMP* columns, and columns with *DEFAULT* constraints.

DEFAULT

Tells the *INSERT* statement simply to create a new record using all of the default values specified for the target table.

```
EXEC[UTE] proc_name { [[@param =] value {[OUTPUT]} [,... ] ] }
```

Tells SQL Server to execute a dynamic Transact-SQL statement, a stored procedure, a Remote Procedure Call (RPC), or an extended stored procedure and store the results in a local table. *Proc_name* is the name of the stored procedure you wish to execute. You may optionally include any of the parameters of the stored procedure, as identified by *@param* (the at sign is required), assign a *value* to the parameter, and optionally designate the parameter as an *OUTPUT* parameter. The columns returned by the result set must match the datatype of the columns in the target table.

Although SQL Server automatically assigns values to *IDENTITY* columns and *TIMESTAMP* columns, it does not do so for *UNIQUEIDENTIFIER* columns. With either of the former datatype columns, you may simply skip them in the *columns* and *values* lists. However, you cannot do that with a *UNIQUEIDENTIFIER* column. You must use the *NEWID*() function to obtain and insert a globally unique ID (GUID):

```
INSERT INTO guid_sample (global_ID, sample_text, sample_int)
```

```
VALUES (NEWID( ), 'insert first record','10000')
```

```
GO
```

When migrating between platforms, remember that inserting an empty string (") into a SQL Server *TEXT* or *VARCHAR* column results in a zero-length string. This is not the same as a null value, as some platforms interpret it.

An example of the *INSERT...EXEC* statements, below, first creates a temporary table called **#ins_exec_container**. Then, the *INSERT* operation retrieves a listing of the *c:\temp* directory and stores it in the temporary table, while the second *INSERT* executes a dynamic *SELECT* statement:

```
CREATE TABLE #ins_exec_container (result_text
```

```
    VARCHAR(300) NULL)
```

```
GO
```

```
INSERT INTO #ins_exec_container
```

```
EXEC master..xp_cmdshell "dir c:\temp"
```

```
GO
```

```
INSERT INTO sales
```

```
EXECUTE ('SELECT * FROM sales_2002_Q4')
```

```
GO
```

This functionality can be very useful in stored procedures where you wish to build business logic using Transact-SQL stored procedures, for example, to determine the state of objects in the database or outside of the database, and then act on those results using Transact-SQL.



SQL Server allows the use of hints to circumvent default query optimization for *INSERT* statements. This type of tuning is only recommended for the most advanced users. Refer to the vendor documentation for more details on hints that are usable with *INSERT*.

INTERSECT Set Operator

The *INTERSECT* set operator retrieves the rows of two or more queries where the rows of the result sets are identical in

both the first and second (and possibly more) queries. In some ways, *INTERSECT* is a lot like an *INNER JOIN* operation.

INTERSECT is in a class of keywords called *set operators*. Other set operators include *EXCEPT* and *UNION*. All set operators are used to simultaneously manipulate the result set of two or more queries; hence the term "set operators."

Platform	Command
DB2	Supported with limitations
MySQL	Not supported
Oracle	Supported with limitations
PostgreSQL	Supported with limitations
SQL Server	Not supported

SQL2003 Syntax

There is technically no limit to the number of queries that you may combine with the *INTERSECT* set operator. The general syntax is:

```
<SELECT statement1>  
  
INTERSECT [ALL | DISTINCT]  
  
[CORRESPONDING [BY (column1, column2, ...)] ]  
  
<SELECT statement2>  
  
INTERSECT [ALL | DISTINCT]  
  
[CORRESPONDING [BY (column1, column2, ...)] ]  
  
...
```

Keywords

ALL

Includes duplicate rows from all result sets.

DISTINCT

Drops duplicate rows from all result sets prior to the *INTERSECT* comparison. Columns containing a NULL value are considered duplicates. (If neither *ALL* nor *DISTINCT* are used, *DISTINCT* behavior is the default.)

CORRESPONDING

Specifies that only columns with the same name in both queries are returned, even though both queries use the asterisk shortcut.

BY

Specifies that only the named columns are returned, even if more columns with corresponding names exist in the queries. Must be used with the *CORRESPONDING* keyword.

Rules at a Glance

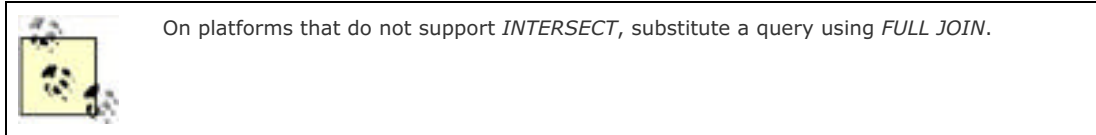
There is only one significant rule to remember when using *INTERSECT*:

The order and number of columns must be the same in all queries. The datatypes of corresponding columns must also be compatible.

The datatypes do not have to be identical, but they should be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest of each column in each ordinal position.

Programming Tips and Gotchas

None of the platforms support the ANSI *CORRESPONDING* [*BY column1, column2, ...*] clause.



The ANSI standard evaluates *INTERSECT* as higher priority than other set operators, while some platforms vary in how they evaluate set operator precedence. You can explicitly control the precedence of set operators using parentheses. Otherwise, the DBMS might evaluate them from leftmost to rightmost expression or from first to last expression.

According to the standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. In order to avoid column and table ambiguity, be sure to alias each column of each table with the same respective alias. For example:

```
SELECT a.au_lname AS 'lastname', a.au_fname AS 'firstname'

FROM authors AS a

INTERSECT

SELECT e.emp_lname AS 'lastname', e.emp_fname AS 'firstname'

FROM employees AS e

ORDER BY lastname, firstname
```

Also, while your column datatypes may be compatible throughout the queries in the *INTERSECT*, there may be some variation in behavior across the DBMS platforms with regard to varying length of the columns. For example, if the **au_lname** column in the first query is markedly longer than the **emp_lname** column in the second query, each platform may apply different rules as to which length is used for the final result. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the columns across the tables have different names. In general, the column names from the first query are used.

DB2

DB2 supports the *INTERSECT* and *INTERSECT ALL* keywords of the ANSI standard, plus the *VALUES* extension:

```
{SELECT statement1 | VALUES (expr1 [,...]) }
```

```
INTERSECT [ALL]
```

```
[CORRESPONDING [BY (column1, column2, ...)] ]
```

```
{SELECT statement2 | VALUES (expr2 [,...]) }
```

```
INTERSECT [ALL]
```

```
[CORRESPONDING [BY (column1, column2, ...)] ]
```

...

Although ANSI's *INTERSECT DISTINCT* is not supported, *INTERSECT* is the functional equivalent. The *CORRESPONDING* clause is not supported.

When using *INTERSECT* (but not *INTERSECT ALL*), the datatypes *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK* and structured types are not usable.

When the same column name is used across queries, then that name is used in the final result set. If, however, the queries use different names for a column, then DB2 generates a new column name for the final result. That column is then unusable in the *ORDER BY* and *FOR UPDATE* clauses.

If multiple set operations are used in a single query, those enclosed in parentheses are evaluated first. Then, the queries are evaluated in left-to-right order. However, all *INTERSECT* operations are performed before any *UNION* and *EXCEPT* operations. For example:

```
SELECT empno
FROM employee
WHERE workdept LIKE 'E%'
INTERSECT
(SELECT empno
 FROM emp_act
 WHERE projno IN ('IF1000', 'IF2000', 'AD3110'))
UNION
VALUES ('AA0001'), ('AB0002'), ('AC0003')
```

The example above retrieves all employee IDs from the **employee** table for employees who are in any department that starts with "E." However, employee IDs are returned only if they also exist in the employee accounts table, named **emp_act**, or have projects of **IF1000**, **IF2000**, or **AD3110**.

MySQL

Not supported.

Oracle

Oracle supports the *INTERSECT* and *INTERSECT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
INTERSECT
<SELECT statement2>
INTERSECT
...
```

Oracle does not support the *CORRESPONDING* clause. The *INTERSECT DISTINCT* is not supported, but *INTERSECT* is the functional equivalent. Oracle *does not* support *INTERSECT* on queries under the following circumstances:

- Queries with columns of *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatypes
- Queries contain *FOR UPDATE* clause or *TABLE* collection expressions

If the first query in the set operation contains any expressions in the select item list, then include the *AS* keyword to associate an alias with the column resulting from the expression. Also, only the first query in the set operation may contain an *ORDER BY* clause.

For example, you could find all store IDs that also have sales:

```
SELECT stor_id FROM stores
INTERSECT
SELECT stor_id FROM sales
```

PostgreSQL

PostgreSQL supports the *INTERSECT* and *INTERSECT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
```

```
INTERSECT [ALL]
```

```
<SELECT statement2>
```

```
INTERSECT [ALL]
```

```
...
```

PostgreSQL *does not* support *INTERSECT* or *INTERSECT ALL* on queries with a *FOR UPDATE* clause. PostgreSQL does not support the *CORRESPONDING* clause. The *INTERSECT DISTINCT* is not supported, but *INTERSECT* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *INTERSECT* or *INTERSECT ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the rightmost occurrence of *ORDER BY* or *LIMIT* will be assumed to apply to the entire set operation. For example, you could find all authors who are also employees and whose last last name starts with "P":

```
SELECT a.au_lname
```

```
FROM authors AS a
```

```
WHERE a.au_lname LIKE 'P%'
```

```
INTERSECT
```

```
SELECT e.lname
```

```
FROM employee AS e
```

```
WHERE e.lname LIKE 'W%';
```

SQL Server

Not supported. However, you can use *NOT IN* or *NOT EXISTS* operations in conjunction with a correlated subquery as alternatives. Refer to the [EXCEPT Set Operator](#) section for examples.

See also

EXCEPT
SELECT
UNION

IS Operator

The *IS* operator determines whether a value is NULL or not.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

{WHERE | {AND | OR} } *expression* IS [NOT] NULL

Keywords

WHERE | {AND | OR} *expression* IS NULL

Returns a Boolean value of TRUE if the *expression* is NULL, and FALSE if the *expression* is not NULL. The *expression* evaluated for NULL can be preceded by the *WHERE* keyword or *AND* or *OR* keywords.

NOT

Inverses the predicate. The statement will instead return a Boolean TRUE if the value of *expression* is not NULL, and FALSE if the value of *expression* is NULL.

Rules at a Glance

Because the value of NULL is unknown, you cannot use comparison expressions to determine if a value is NULL. For example, the expressions *X = NULL* and *X <> NULL* cannot be resolved because no value can equal, or not equal, an unknown.

Instead, you must use the *IS NULL* operator. Be sure that you do not put the word NULL within quote marks, because if you do that, the DBMS will interpret the value as the word "NULL" and not the special value of NULL.

Programming Tips and Gotchas

Some platforms support the use of a comparison operator to determine if an expression is NULL. However, all platforms covered by this book now support the ANSI *IS [NOT] NULL* syntax.

Sometimes, checking for NULL will make your *WHERE* clause only slightly more complex. For example, rather than a simple predicate to test the value of **stor_id**:

```
SELECT stor_id, ord_date
FROM sales
WHERE stor_id IN (6630, 7708)
```

you need to add a second predicate to accommodate the possibility that **stor_id** might be NULL:

```
SELECT stor_id, ord_date
FROM sales
WHERE stor_id IN (6630, 7708)
OR stor_id IS NULL
```

See Also

```
SELECT
WHERE
```

JOIN Subclause

The *JOIN* subclause enables you to retrieve rows from two or more logically related tables. You are able to define many different join conditions and types of joins, though the types of joins supported by the different platforms vary greatly.

Platform	Command
DB2	Supported, with limitations
MySQL	Supported, with variations
Oracle	Supported
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL2003 Syntax

```
FROM table [AS alias] {CROSS JOIN |  
{ [NATURAL] [join_type] JOIN joined_table [AS alias]  
  { ON join_condition1 [{AND|OR} join_condition2] [...] } |  
  USING (column1 [,...]) } }  
[...]
```

Keywords

FROM *table*

Defines the first table or view in the join.

NATURAL

Specifies that the join (either inner or outer) should be assumed on the tables using all columns of identical name shared between the two tables. Consequently, you should *not* specify join conditions using the *ON* or *USING* clauses. The query will fail if you issue a natural join on two tables that do not contain any columns with the same name(s).

join_type JOIN *joined_table*

Specifies the type of *JOIN* and the second (and any subsequent) table in the join. You may also define an *alias* on any of the *joined_tables*. The join types are:

CROSS JOIN

Specifies the complete cross product of two tables. For each record in the first table, all the records in the second table are joined, creating a *huge* result set. This command has the same effect as leaving off the join condition, and is also known as a "Cartesian Product." *Cross joins are not advisable or recommended.*

[*INNER*] *JOIN*

Specifies that unmatched rows in either table of the join should be discarded. If no join type is explicitly defined in the ANSI style, then this is the default.

LEFT [*OUTER*] *JOIN*

Specifies that all records be returned from the table on the left side of the join statement. If a record returned from the left table has no matching record in the table on the right side of the join, it is still returned. Columns from the right table return NULL values when there is no matching row. It is a good idea to configure all your outer joins as left outer joins, rather than mixing left and right outer joins, wherever possible for consistency.

RIGHT [OUTER] JOIN

Specifies that all records be returned from the table on the right side of the join statement, even if the table on the left has no matching record. Columns from the left table return NULL values when there is no matching row.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

UNION JOIN

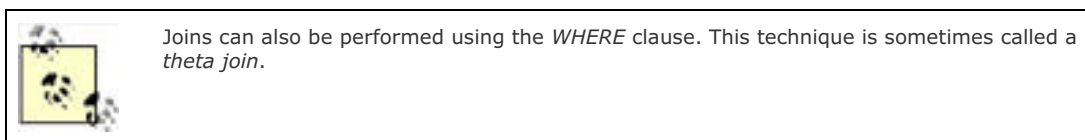
Specifies that all the columns of both tables and every row of both table will be returned. Any columns that have no value in the corresponding joined table are assigned a NULL value.

AS alias

Specifies an alias or shorthand for the joined table. The *AS* keyword is optional when specifying an alias.

ON join_condition

Joins together the rows of the table shown in the *FROM* clause to the rows of the table declared in the *JOIN* clause. You may have multiple *JOIN* statements, all based on a common set of values. These values are usually contained in columns of the same name and datatype appearing in both tables being joined. These columns, or possibly a single column from each table, are called the *join key* or *common key*. Most—but not all—of the time, the join key is the primary key of one table and a foreign key in another table. As long as the data in the columns match, the join can be performed.



Join_conditions are syntactically depicted in the following form (note that join types are intentionally excluded in this example):

FROM table_name1

JOIN table_name2

ON table_name1.column1 = table_name2.column2

[{AND|OR} *table_name1.column3 = table_name2.column4*]

[...]

JOIN table_name3

ON table_name1.columnA = table_name3.columnA

[{AND|OR} *table_name1.column3 = table_name2.column4*]

[...]

[JOIN...]

Use the *AND* operator and the *OR* operator to issue a *JOIN* with multiple conditions. It is also a good idea to use brackets around each pair of joined tables if more than two tables are involved. This makes reading the query much easier.

USING (*column* [,...])

Assumes an equality condition on one or more named *columns* that appear in both tables. The column (or columns) must exist, as named, in both tables. Writing a *USING* clause is a little quicker than writing *...ON table1.columnA = table2.columnA*, but the results are functionally equivalent.

Rules at a Glance

Joins enable you to retrieve records from two (or more) logically related tables in a single result set. You can use an ANSI *JOIN* (detailed here) to perform this operation, or something called a *theta join*. Theta joins, which use the *WHERE* clause to establish the filtering criteria, are the "old" way to do join operations.

For example, you might have a table called **employee** that tracks information about everyone employed in your company. The **employee** table, however, doesn't have extensive information about the job an employee holds. Instead, the **employee** table holds only a **job_id**. All information about the job, such as **description** and **title**, are stored in a table called **job**. Using a *JOIN*, you can easily return columns from both tables in a single set of records. The following sample queries illustrate the difference between a theta and an ANSI *JOIN*:

/ Theta join */*

```
SELECT emp_lname, emp_fname, job_title
```

```
FROM employee, jobs
```

```
WHERE employee.job_id = jobs.job_id;
```

/ ANSI join */*

```
SELECT emp_lname, emp_fname, job_title
```

```
FROM employee
```

```
JOIN jobs ON employee.job_id = jobs.job_id;
```

Whenever you reference multiple columns in a single query, the columns must be unambiguous. In other words, the columns must either be unique to each table, or they must be referenced with a table identifier. In the example above, both tables have a **job_id** column, so references to **job_id** must be qualified with a table identifier. The columns in the query that don't exist in both tables don't need to be qualified by table identifiers. However, queries like this are often very hard to read. The following variation of the previous ANSI join is in better form, because it uses the short, and easy to read, alias **e** to refer to the table:

```
SELECT e.emp_lname, e.emp_fname, j.job_title
```

```
FROM employee AS e
```

```
JOIN jobs AS j ON e.job_id = j.job_id;
```

Previous examples were limited to equi-joins — joins using equality and an equal sign (=). However, most other comparison operators are allowed. You can perform joins on *>*, *<*, *>=*, *<=*, *<>*, and so forth.

You cannot join on large object binary datatypes (i.e., *BLOB*) or other large object datatypes (e.g., *CLOB*, *NLOB*, etc.). Other datatypes are usually allowed in a join comparison.

Following are examples of each type of join:

CROSS JOIN

Following are some cross join examples. The first is a theta join that simply leaves off the join conditions. The second is written using the *CROSS JOIN* clause. The final query is similar in concept to the first, with a *JOIN* clause that omits the join conditions.

```
SELECT *  
  
FROM employee, jobs;
```

```
SELECT *  
  
FROM employee  
CROSS JOIN jobs;
```

```
SELECT *  
  
FROM employee  
JOIN jobs;
```

INNER JOIN

Following is an inner join written using the newer syntax:

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
INNER JOIN publishers AS p ON a.city = p.city  
ORDER BY a.au_lname DESC
```

There are lots of authors in the **authors** table, but very few of them have a city that matches the city of the publishers in the **publishers** table. For example, the above query executed in the **pubs** database on a SQL Server produces results like the following:

first name	last name	publisher
Carson	Cheryl	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems

The join is called an *inner join* because only those records that meet the join condition in both tables are said to be "inside" the join. You could also issue the same query, on platforms that support it, by substituting the *USING* clause for the *ON* clause:

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
INNER JOIN publishers AS p USING (city)  
ORDER BY a.au_lname DESC
```

The results for the above query would be the same.

LEFT [OUTER] JOIN

It is a good idea to configure all your outer joins as left outer joins, for greater consistency, rather than mixing left and right outer joins.

In the following example, we can show a *LEFT OUTER JOIN* by asking for the publisher for each author (we could also substitute the *USING* clause for the *ON* clause, as shown in the example for *INNER JOIN* shown earlier):

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
LEFT OUTER JOIN publishers AS p ON a.city = p.city  
ORDER BY a.au_lname DESC
```

In the above example, every author will be returned along with the publisher's name, where there is a match, or a NULL value, where there is no match. For example, in the SQL Server **pubs** database, the query above returns:

first name	last name	publisher
Yokomoto	Akiko	NULL
White	Johnson	NULL
Stringer	Dirk	NULL
Straight	Dean	NULL
...		

All of the data is returned in the lefthand columns (those from the **authors** table) and any column on the righthand side (those from the **publishers** table) that does not have a match returns a NULL value. The result set shows NULL values where no data matches in the join.

RIGHT [OUTER] JOIN

RIGHT OUTER JOIN is essentially the same as a *LEFT OUTER JOIN*, except everything is weighted toward the table on the right side of the query. For example, the following query executed in the **pubs** database on a SQL Server:

```
SELECT a.au_lname AS 'first name',  
       a.au_fname AS 'last name',  
       p.pub_name AS 'publisher'  
FROM authors AS a  
RIGHT OUTER JOIN publishers AS p ON a.city = p.city  
ORDER BY a.au_lname DESC
```

returns the following result set:

first name	last name	publisher
Carson	Cheryl	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems
NULL	NULL	New Moon Books
NULL	NULL	Binnet & Hardley

All of the data is returned on the righthand side of the query (hence the term "right outer join") and any columns from the left-hand side that do not have a match return a NULL value. The result set shows NULL values where no data matches in the join.

NATURAL [INNER | {LEFT | RIGHT}[OUTER]] JOIN

Natural joins are a substitute for the *ON* or *USING* clause, so do not use *NATURAL* with the *ON* or *USING* clauses. For example:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors AS a
NATURAL RIGHT OUTER JOIN publishers AS p
ORDER BY a.au_lname DESC
```

The preceding query will work as the earlier examples, but only if both tables possess a column called **city** and that is the only column that they hold in common. You could similarly perform any of the other types of joins (*INNER*, *FULL*, *OUTER*) using the *NATURAL* prefix.

FULL [OUTER] JOIN

Specifies that all rows from either table be returned, regardless of matching records in the other table. The result set shows NULL values, where no data matches in the join. Note that the *OUTER* keyword is optional. If we take our earlier example query and render it as a *FULL JOIN*, it looks like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors AS a
FULL JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

The result set returned by the query is actually the accumulation of the result sets of issuing separate *INNER*, *LEFT*, and *RIGHT* join queries (some records have been excluded for brevity):

first name	last name	publisher
Yokomoto	Akiko	NULL
White	Johnson	NULL
Stringer	Dirk	NULL

...

Dull	Ann	NULL
del Castillo	Innes	NULL
DeFrance	Michel	NULL
Carson	Cheryl	Algodata Infosystems
Blotchet-Halls	Reginald	NULL
Bennet	Abraham	Algodata Infosystems
NULL	NULL	Binnet & Hardley
NULL	NULL	Five Lakes Publishin
NULL	NULL	New Moon Books
...		
NULL	NULL	Scootney Books
NULL	NULL	Ramona Publishers
NULL	NULL	GGG&G

As you can see above, with the *FULL JOIN* you get some records with the NULLs on the right and data on the left (*LEFT JOIN*), some with all data (*INNER JOIN*), and some with NULLs on the left and data on the right (*RIGHT JOIN*).

Programming Tips and Gotchas

If an explicit *join_type* is omitted, then an *INNER JOIN* is assumed. Note that there are many types of joins, each with their own rules and behaviors, described in the "Rules at a Glance."

In general, you should favor the *JOIN* clause over the *WHERE* clause for describing join expressions. This not only keeps your code cleaner, making it easy to differentiate join conditions from search conditions, but you also avoid the possibility of buggy behavior resulting from some platform-specific implementations of outer joins specified using the *WHERE* clause.

In general, we do not recommend the use of labor-saving keywords like *NATURAL*, since the subclause will not automatically update itself when the structure of the underlying tables change. Consequently, statements using these constructs may fail when a table change is introduced without also changing the query.

Not all join types are supported by all platforms, so refer to the sections below for full details on platform-specific join support.



Joins involving more than two tables can be difficult. When joins involve three or more tables, it is a good idea to think of the query as a series of two table joins.

DB2

DB2 supports *INNER* and *OUTER* joins using the *ON* condition clause. It does not support *CROSS* or *NATURAL* join syntax, nor the *USING* clause. DB2's JOIN clause syntax is:

```
FROM table [AS alias]
```

```
{ {[INNER] | [ {LEFT | RIGHT | FULL} [OUTER]] }
```

```
JOIN joined_table [AS alias]
```

```
{ ON join_condition1 [{AND|OR} join_condition2 [ ... ] }
```

```
[...]
```


Refer to the "Rules at a Glance" section for examples.

MySQL

MySQL supports most ANSI syntax, except that natural joins are only supported on outer joins, not on inner joins. MySQL's *JOIN* syntax is:

```
FROM table [AS alias]
{[STRAIGHT_JOIN joined_table] |
{ {[INNER] | | [CROSS] |
[NATURAL] [ {LEFT | RIGHT | FULL} [OUTER]]}
JOIN joined_table [AS alias]
{ ON join_condition1 [{AND|OR} join_condition2] [...] ] |
USING (column1 [,...]) } } }
[...]
```

where:

STRAIGHT_JOIN

Forces the optimizer to join tables in the exact order they appear in the *FROM* clause.

The *STRAIGHT_JOIN* keyword is functionally equivalent to *JOIN*, except that it forces the join order from left to right. This option was supplied because MySQL might, rarely, join the tables in the wrong order.

Refer to the [Rules at a Glance](#) section for examples.

Oracle

Oracle fully supports the entire ANSI standard *JOIN* syntax. However, Oracle only began to support ANSI standard *JOIN* syntax in Version 9. Consequently, any older Oracle SQL code exclusively uses *WHERE* clause joins. Oracle's old syntax for outer theta joins included adding (+) to column names on the side *opposite* of the outer table. Oracle's old syntax for outer joins comes from the fact that the table supplying the null value rows in effect has null value rows added to it. Oracle represents this by appending the expression "(+)" to the column name on the opposite side of the direction of the join.

For example, the following query does a *RIGHT OUTER JOIN* on the **authors** and **publishers** tables. The old Oracle syntax looks like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors a, publishers p
WHERE a.city(+) = p.city
ORDER BY a.au_lname DESC
```

while the same query in the ANSI syntax would look like:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
FROM authors AS a
RIGHT OUTER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

Refer to the [Rules at a Glance](#) section for more *JOIN* examples.

Oracle is unique in offering partitioned outer joins, which are useful for filling gaps in result sets due to sparse data storage. For example, assume we store production records in a manufacturing table keyed on day and **product_id**. The table holds a row showing quantity produced for each product for any day on which it is made, but there are no rows for the days it is not produced. This is considered sparse data, since a list of all rows will not show every day for every product. For calculation and reporting purposes, it's very useful to create result sets where each product has a row for every day, whether or not it was manufactured. Partition outer join makes it simple to do that, since it lets you define a logical partition and apply an outer join to each partition value. Here is the manufacturing example, which does a partitioned outer join with a **times** table to make sure each **product_id** has the full set of dates in a specified time range:

```
SELECT times.time_id AS time, product_id AS id, quantity AS qty
FROM manufacturing
PARTITION BY (product_id)
RIGHT OUTER JOIN times
ON (manufacturing.time_id = times.time_id)
WHERE manufacturing.time_id
BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
      AND TO_DATE('06/10/05', 'DD/MM/YY')
ORDER BY 2, 1;
```

Following is the output from this query:

time	id	qty
01-OCT-05	101	10
02-OCT-05	101	
03-OCT-05	101	
04-OCT-05	101	17
05-OCT-05	101	23
06-OCT-05	101	
01-OCT-05	102	
02-OCT-05	102	
03-OCT-05	102	43
04-OCT-05	102	99
05-OCT-05	102	
06-OCT-05	102	87

To get these results without using a partitioned outer join would require more complex and less efficient SQL.

PostgreSQL

PostgreSQL fully supports the ANSI standard. Refer to the [Rules at a Glance](#) section for examples.

SQL Server

SQL Server supports *INNER*, *OUTER*, and *CROSS* joins using the *ON* clause. SQL Server does not support *NATURAL* join syntax, nor the *USING* clause. SQL Server's *JOIN* syntax is:

```
FROM table [AS alias]  
  
{ {[INNER] | [CROSS] | [ {LEFT | RIGHT | FULL} [OUTER]] }  
  
  JOIN joined_table [AS alias]  
  
    { ON join_condition1 [{AND|OR}  
  
      join_condition2] [...] }  
  
[...]
```

Refer to the [Rules at a Glance](#) section for examples.

See Also

```
SELECT  
WHERE
```

LIKE Operator

The *LIKE* operator enables specified string patterns in *SELECT*, *INSERT*, *UPDATE*, and *DELETE* statements to be matched, specifically in the *WHERE* clause. A specified pattern may include special wildcard characters. The specific wildcards supported vary from platform to platform.

Platform	Command
DB2	Supported
MySQL	Supported, with variations
Oracle	Supported
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
WHERE expression [NOT] LIKE string_pattern  
  
[ESCAPE escape_sequence]
```

Keywords

WHERE *expression* LIKE

Returns a Boolean TRUE when the value of *expression* matches the *string_pattern*. The expression may be a column, a constant, a host variable, a scalar function, or a concatenation of any of these. It should not be a user-defined type, nor should it be certain types of *LOBs*.

NOT

Inverses the predicate. The statement returns a Boolean TRUE if the value of *expression* does not contain the *string_pattern*, and FALSE if the value of *expression* contains the *string_pattern*.

ESCAPE *escape_sequence*

Allows you to search for the presence of characters that would normally be interpreted as wildcards.

Rules at a Glance

Matching string patterns is easy with *LIKE*, but there are a couple of simple rules to remember:

- All characters, including trailing and leading spaces, are important.
- Differing datatypes may be compared using *LIKE*, but they store string patterns differently. In particular, be aware of the differences between *CHAR*, *VARCHAR*, and *DATE* datatypes.
- Using *LIKE* may negate indexes or force the DBMS to use alternative, less optimal indexes than a straight comparison operation.

The ANSI standard currently supports two wildcard operators that are supported by all of the platforms covered in this book:

%

Matches any string.

_ (*underscore*)

Matches any single character.

The first query in the following example retrieves any city record with "ville" in its name. The second query returns authors with a first name *not* like Sheryl or Cheryl (or Aheryl, Bheryl, Dheryl, Zheryl, and so forth).

```
SELECT * FROM authors
```

```
WHERE city LIKE '%ville%';
```

```
SELECT * FROM authors
```

```
WHERE au_fname NOT LIKE '_heryl';
```

Some of the platforms support additional wildcard symbols. These are described in the platform-specific sections that follow.

Use of the *ESCAPE* clause allows you to look for wildcard characters in the strings stored in your database. Using this mechanism, you designate a character as your escape character, typically a character that does not otherwise appear in the pattern string. For example, you might designate the tilde (~) because you know it never appears in the pattern string. Any wildcard character preceded by the escape sequence is then not treated as a wildcard, but rather as the character itself. For example, to look through the **comments** column of the **sales_detail** table (on SQL Server) to see whether any customers have mentioned a newly introduced discount:

```
SELECT ord_id, comment
```

```
FROM sales_detail
```

```
WHERE comment LIKE '%~%' ESCAPE '~'
```

In this case, the first and last % are interpreted as wildcards. But the second % character is interpreted as just that, a % character, because it is preceded by the designated escape sequence.

Programming Tips and Gotchas

The usefulness of *LIKE* is based on the wildcard operators that it supports. *LIKE* returns a Boolean TRUE value when the comparison finds one or more matching values.

The default case sensitivity of the DBMS is very important to the behavior of *LIKE*. For example, Microsoft SQL Server is not case-sensitive by default (though it can be configured that way). Thus, Microsoft SQL Server will evaluate the strings *DAD* and *dad* to be equal. Oracle, on the other hand, is case-sensitive. Thus, on Oracle, a comparison of *DAD* and *dad* would show them to be unequal. Here's an example query to better illustrate this point:

```
SELECT *  
  
FROM authors  
  
WHERE Iname LIKE 'LARS%'
```

This query on Microsoft SQL Server would find **authors** whose last names are stored as 'larsen' or 'lars,' even though the search was for uppercase 'LARS%'. Oracle would not find 'Larsen' or 'Lars', because Oracle performs case-sensitive comparisons.

DB2

DB2 supports the ANSI SQL2003 syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It supports *ESCAPE* sequences.

DB2 is case sensitive, so its implementation of *LIKE* is completely case sensitive. You should use a function like *UPPER* or *TRANSLATE* to ensure that differently cased values are always evaluated uniformly. In addition, DB2 will implicitly convert the code page of either the *string_pattern* or the *escape_sequence* to that of the *expression*, as long as none of them are defined as *FOR BIT DATA*.

MySQL

MySQL supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It also supports the *ESCAPE* clause.

MySQL also supports the special functions of *REGEXP* and *RLIKE*, plus *NOT REGEXP* and *NOT RLIKE* for the evaluation of regular expressions. MySQL, after Version 3.23.4, is not case sensitive by default.

Oracle

Oracle supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It supports *ESCAPE* sequences. Oracle's *LIKE* syntax is as follows:

```
WHERE expression [NOT] {LIKE | LIKEC | LIKE2 |  
  
    LIKE4} string_pattern  
  
[ESCAPE escape_sequence]
```

The Oracle-specific syntax elements have the following meanings:

LIKEC

Uses UNICODE complete characters.

LIKE2

Uses UNICODE USC2 codepoints.

LIKE4

Uses UNICODE UCS4 codepoints.

Since Oracle is case sensitive, you should enclose the *expression*, the *string_pattern*, or both, with the *UPPER* function. That way, you are always comparing apples to apples.

PostgreSQL

PostgreSQL supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards. It supports *ESCAPE* sequences.

PostgreSQL is case sensitive by default. PostgreSQL provides the keyword *ILIKE* for case-insensitive pattern matching. You can also use the operator *~~* as an equivalent to *LIKE*, and *~~** for *ILIKE*, as well as *!~~* and *!~~** for *NOT LIKE* and *NOT ILIKE*, respectively. These are all extensions to the ANSI SQL2003 standard.

For example, the following queries are functionally the same:

```
SELECT * FROM authors
WHERE city LIKE '%ville';

SELECT * FROM authors
WHERE city ~~ '%ville';
```

Since these queries are in lowercase, you might run into a case-sensitivity problem. That is, the queries are looking for a lowercase '%ville', but the table might contain uppercase (and unequal) values of 'BROWNSVILLE', 'NASHVILLE', and 'HUNTSVILLE'. You can get around this as shown below:

```
-- convert the values to upper case

SELECT * FROM authors
WHERE city LIKE UPPER('%ville');

-- perform the pattern match using case insensitivity

SELECT * FROM authors
WHERE city ~~* '%ville';

SELECT * FROM authors
WHERE city ILIKE '%ville';
```

Although beyond the scope of this text, you should be aware that PostgreSQL also supports POSIX regular expressions. See the platform documentation for details.

SQL Server

SQL Server supports the ANSI syntax for *LIKE*. It supports *ESCAPE* sequences. It supports the following additional wildcard operators:

[]

Matches any value in the specified set, as in [abc], or any range, as in [k-n].

[^]

Matches any characters not in the specified set or range.

Using SQL Server's additional wildcard operators, you have some added capabilities. For example, you can retrieve any author with a last name like Carson, Carsen, Karson, or Karsen:

```
SELECT * FROM authors
WHERE au_lname LIKE '[CK]ars[eo]n'
```

Or you can retrieve any author with a last name that ends in "arson" or "arsen," *but not* Larsen or Larson:

SELECT * FROM authors

WHERE au_lname LIKE '[A-Z^L]ars[eo]n'



Remember that when performing string comparisons with *LIKE*, all characters in the pattern string are significant, including all leading and trailing blank spaces.

See Also

SELECT, UPDATE, DELETE
WHERE

MERGE Statement

The *MERGE* statement is sort of like a *CASE* statement for DML operations. It combines *UPDATE* and *INSERT* statements into a single atomic statement with either/or functionality.

Basically, *MERGE* examines the records of a source table and a target table. If the records exist in both the source and target tables, based upon predefined conditions, then records in the target table are updated with the values of the records in the source table. If records do not exist in the target table that do exist in the source table, then they are inserted into the target table. The *MERGE* statement was added in the SQL2003 release of the ANSI standard.

Platform	Command
DB2	Supported
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

MERGE INTO {*object_name* | *subquery*} [[AS] *alias*]

USING *table_reference* [[AS] *alias*]

ON *search_condition*

WHEN MATCHED

THEN UPDATE SET *column* = { *expression* | DEFAULT } [, ...]

WHEN NOT MATCHED

THEN INSERT [(*column* [, ...])] VALUES (*expression* [, ...])

Keywords

MERGE INTO {*object_name* | *subquery*}

Declares the target object of the merge operation. The target object may be a table or updateable view of *object_name*, or the target object may be a nested table subquery rather than an explicitly declared table or updateable view.

[AS] *alias*

Provides an optional alias for the target table.

USING *table_reference*

Declares the source table, view, or subquery of the merge operation.

ON *search_condition*

Specifies the condition or conditions on which a match between the source and target table are evaluated. The syntax is essentially the same as the *ON* subclause of the *JOIN* clause. For example, when merging records from the **new_hire_emp** table into the **emp** table the clause might look like *ON emp.emp_id = new_hire_emp.emp_id*.

WHEN MATCHED THEN UPDATE SET *column* = { *expression* | DEFAULT } [, ...]

Declares that if a record from the source table has a matching record in the target table (based on the *search_condition*), then one or more specified *columns* of the target table are updated with the indicated value of *expression*.

WHEN NOT MATCHED THEN INSERT [(*column* [, ...])] VALUES (*expression* [, ...]

Declares that if a record from the source table does not have a matching record in the target table (based on the *search_condition*), then a new record is inserted into the target table using one or more specified *columns* using the value of *expression*.

Rules at a Glance

The rules for using *MERGE* are straightforward:

- The *WHEN MATCHED* and *WHEN NOT MATCHED* clauses are required, but may not be specified more than once.
- The target table can be a standard updateable table, an updateable view, or an updateable subquery.
- If the *table_reference* is a subquery, enclose it in parentheses.
- The *search_condition* clause should not contain any references to stored procedures or user-defined functions.
- The *search_condition* clause may contain multiple elements using the *AND* or *OR* operators.
- If the column list is omitted from the *WHEN NOT MATCHED* clause, then a column list of all the columns in the target table, in ordinal position, is assumed.
- Other important rules used by the *MERGE* statement are self-evident. For example, the columns referenced in the *WHEN MATCHED* clause must be updateable.

Programming Tips and Gotchas

The *MERGE* statement has sometimes been nicknamed the "upsert" statement. That is because it allows, in a single operation, a set of records to be either inserted into a table or, if they already exist, updated with new values.

The only tricky aspect of the *MERGE* statement is simply getting used to the idea of either/or processing of the *INSERT* and *UPDATE* statements.

Assume that we have two tables. The **EMP** table contains all employees of the company who have successfully completed the mandatory 90-day probationary period at the start of their employment. Employees in the **EMP** table can also have several statuses, such as active, inactive, and terminated. Any new hire to the company is recorded in the **NEW_HIRE** table. After 90 days, they are moved into the **EMP** table like all other regular employees. However, since

our company hires college interns every summer, it's very likely that some of our new hires will actually have a record in the **EMP** table from last year with a status of inactive. Using pseudocode, the business problem is summarized as:

```
For each record in the NEW_HIRE table
    Find the corresponding record in the EMP table
    If the record exists in the EMP table
        Update existing data in the EMP table
    Else
        Insert this record into the EMP table
    End If
End For
```

We could write a rather lengthy stored procedure that will examine all the records of the **NEW_HIRE** table and then conditionally perform *INSERT* statements for the entirely new employees or *UPDATE* statements for the returning college interns. But the following ANSI standard *MERGE* statement makes this process much easier:

```
MERGE INTO emp AS e
    USING (SELECT * FROM new_hire) AS n
    ON e.empno = n.empno
WHEN MATCHED THEN
    UPDATE SET
        e.ename = n.ename,
        e.sal   = n.sal,
        e.mgr   = n.mgr,
        e.deptno = n.deptno
WHEN NOT MATCHED THEN
    INSERT ( e.empno, e.ename, e.sal, e.mgr, e.deptno )
    VALUES ( n.empno, n.ename, n.sal, n.mgr, n.deptno );
```

As you can see, the *MERGE* statement is very useful for data loading operations.

DB2

DB2 supports the ANSI standard for the *MERGE* statement.

MySQL

Not supported.

Oracle

Oracle supports the *MERGE* statement with only the tiniest variations, which are almost entirely evident by comparing the Oracle syntax diagram against the ANSI standard syntax diagram:

```
MERGE INTO [schema.] {object_name | subquery} [alias ]
USING [schema.]table_reference [alias]
ON ( search_condition )
```

WHEN MATCHED THEN

```
UPDATE SET column = { expression | DEFAULT } [, ...]
```

WHEN NOT MATCHED THEN

```
INSERT ( column [, ...] ) VALUES ( expression [, ...] )
```

The differences between the ANSI standard and Oracle's implementation include:

- The Oracle implementation does not allow the *AS* keyword with assigning an alias to the target or source table.
- Oracle requires parentheses around the *search_condition* clause.
- The *WHEN NOT MATCHED* clause requires an insert column list, while the ANSI standard makes it optional.

Refer to the examples in the earlier "Rules at a Glance" and "Programming Tips and Gotchas" for more information.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

INSERT
JOIN
SELECT
SUBQUERY
UPDATE

OPEN Statement

The *OPEN* statement is one of four commands used in cursor processing, along with *DECLARE*, *FETCH*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than in a complete set. The *OPEN* statement opens a pre-existing server cursor created with the *DECLARE CURSOR* statement.

Cursors are especially important on relational databases because databases are set-based, while most client-centric programming languages are row-based. So cursors allow programmers and databases to perform operations a single row at a time, while the default behavior of a relational database is to operate on a whole set of records.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported

SQL2003 Syntax

OPEN cursor_name

Keywords

OPEN cursor_name

Identifies and opens the previously defined cursor created with the *DECLARE CURSOR* command.

Rules at a Glance

At the highest level, a cursor must be:

1. Created using *DECLARE*
2. Opened using *OPEN*
3. Operated against using *FETCH*
4. Dismissed using *CLOSE*

By following these steps, you create a result set similar to that of a *SELECT* statement, except that you can operate against each individual row within the result set. For example, assume, on a DB2 database, that you've created a cursor called **employee_cursor**, containing three columns:

```
DECLARE CURSOR employee_cursor FOR
```

```
SELECT lname, fname, emp_id
```

```
FROM employee
```

```
WHERE hire_date >= 'FEB-14-2004';
```

Having created the cursor, you can now open the cursor and fetch data from it into some host variables:

```
OPEN employee_cursor;
```

```
FETCH NEXT FROM employee_cursor
```

```
INTO :emp_last_name, :emp_first_name, :emp_id
```

The *FETCH* statement allows you to position the current row within the result set retrieved by the the cursor. If you haven't accessed the cursor yet, you will be given the first record in the result set. You can then use any procedural code you want to operate against the values in that row.

The following generic SQL example opens a cursor and fetches all the author firstname and author lastname values from the **authors** table:

```
DECLARE employee_cursor CURSOR FOR
```

```
SELECT au_lname, au_fname
```

```
FROM pubs.dbo.authors
```

```
WHERE lname LIKE 'K%'
```

```
OPEN employee_cursor
```

```
FETCH NEXT FROM employee_cursor
```

```
BEGIN
```

```
    FETCH NEXT FROM employee_Cursor
```

```
END
```

```
CLOSE employee_cursor
```

Programming Tips and Gotchas

The most common error encountered with the *OPEN* statement is failing to close the cursor properly. Although the *OPEN* statement is detailed in isolation here, it should always be managed as a group with the *DECLARE*, *FETCH*, and *CLOSE* statements. Although you will not get an error message for failing to close a cursor, the cursor may continue to consume memory, locks, and other resources on the server as long as it is open. For example, if you forget to close cursors, you could be creating a headache somewhat similar to a memory leak — each cursor consumes memory until it is closed. Even if you no longer use a cursor, it is still taking up memory that the database server might otherwise be using elsewhere. So take a little extra time to make sure that every declared and opened cursor is eventually closed.

Cursors are often used in stored procedures and in batches of procedural code. They are useful when you need to perform actions on individual rows rather than on entire sets of data at a time. But because cursors operate on individual rows and not on sets of data, they are often much slower than other means of accessing data. Just make sure that you perform strong analysis on your approach. Many challenges, such as a convoluted *DELETE* operation or a very complex *UPDATE*, can be solved by using clever *WHERE* and *JOIN* clauses instead of cursors.

DB2

DB2 supports the ANSI standard for *OPEN* with a couple of additions:

```
OPEN cursor_name [USING {variable_name1 [,...]} | DESCRIPTOR descriptor_name]
```

The DB2 specific syntax elements are:

```
USING {variable_name1 [,...]} | DESCRIPTOR descriptor_name
```

Supplies a list of variables for any parameters in the prepared *SELECT* statement. If the *DECLARE CURSOR* statement used a prepared *SELECT* statement with variables, you cannot use the *USING* clause. (If you use *USING* and there are no parameters, then *USING* is ignored.) The *USING* clause elements are as follows:

```
variable_name1 [,...]
```

Specifies a host variable or variables. As you would expect, each variable corresponds to a parameter of the same ordinal position. DB2 locator variables and file reference variables are allowable, in addition to host variables. Host variables are used for static SQL.

```
DESCRIPTOR descriptor_name
```

Identifies a descriptor area (SQLDA) containing a valid description of the dynamic SQL variable(s). Descriptor variables are used for dynamic SQL.

For example, the following DB2 code declares several host (called **hostvarxxx**) and dynamic variables (defined with the *PREPARE* statement), and then uses them in a cursor called **dynamic_cursor**:

```
EXEC SQL BEGIN DECLARE SECTION;  
  
    static short hostvar_int;  
  
    char hostvar_vchar64[65];  
  
    char statement_str[200];  
  
EXEC SQL END DECLARE SECTION;  
  
EXEC SQL PREPARE statement_name  
    FROM :statement_str;  
  
EXEC SQL DECLARE dynamic_cursor CURSOR  
    FOR statement_name;  
  
EXEC SQL OPEN dynamic_cursor  
    USING :hostvar_int, :hostvar_vchar64;
```

MySQL

Not supported.

Oracle

Oracle fully supports the ANSI standard. Plus, Oracle allows parameters to be passed directly into the cursor when it is opened. Do this using the following format:

```
OPEN cursor_name parameter1 [...]
```

PostgreSQL

PostgreSQL does not have an *OPEN CURSOR* statement. Instead, PostgreSQL implicitly opens a cursor when it is created by the *DECLARE* statement.

SQL Server

In addition to the standard *OPEN* statement, SQL Server allows global cursors using the following syntax:

```
OPEN [GLOBAL] cursor_name
```

where:

cursor_name

Specifies the name of the cursor, or is a string variable containing the name of the cursor, created earlier with the *DECLARE CURSOR* statement.

GLOBAL

Enables the cursor to be referenced by multiple users, even if they have not explicitly been assigned permissions to the cursor. Otherwise, a local cursor is assumed.

SQL Server allows you to declare several different kinds of cursors. If a cursor is *INSENSITIVE* or *STATIC*, then the *OPEN* statement creates a temporary table to hold the result set of the cursor. Similarly, if the declared cursor has the *KEYSET* option, then a temporary table is automatically created to hold the keyset.

See Also

CLOSE CURSOR
DECLARE CURSOR
FETCH
SELECT

ORDER BY Clause

The *ORDER BY* clause specifies the sort order of the result set retrieved by a *SELECT* statement.

Platform	Command
DB2	Supported, with limitations
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL2003 Syntax

`ORDER BY {sort_expression [COLLATE collation_name]
[ASC | DESC]} [,...]`

Keywords

ORDER BY

Specifies the order in which rows should be returned by a query. You should not anticipate a specific ordering if you exclude the *ORDER BY* clause, even if you specify a *GROUP BY* clause and it appears that a sort has been done.

sort_expression

Specifies an item in the query that will help determine the order of the result set. You can have multiple *sort_expressions*, and they are usually column names or column aliases from the query. However, they may also be expressions like (*salary * 1.02*). SQL92 allowed the use of ordinal positions for *sort_expressions*. This functionality has been deprecated, and should not be used in SQL2003 queries.

COLLATE *collation_name*

Overrides the default collation of the *sort_expression*, and applies the *collation_name* to the sort expression for the purposes of the *ORDER BY* clause.

[ASC | DESC]

Specifies that the *sort_expression* should be returned in either ascending order (*ASC*) or descending order (*DESC*).

Rules at a Glance

The *ORDER BY* clause should reference columns as they appear in the select item list of the *SELECT* statement, preferably using their alias, if one exists. For example:

```
SELECT au_fname AS first_name, au_lname AS last_name  
FROM authors  
ORDER BY first_name, last_name
```

The *ORDER BY* clause uses a major-to-minor sort ordering. This means that the result set is ordered by the first column referenced, equal values in the first column are then ordered by the second column, equal values in the second column are then ordered by the third column, and so forth.

The individual aspects of a column's ordering — *COLLATE* and *ASC/DESC* — are independent of the other columns in the *ORDER BY* clause. Thus, you could order a result set in ascending order by one column, and then flip the next column and order it in descending order:

```
SELECT au_fname AS first_name, au_lname AS last_name  
FROM authors  
ORDER BY au_lname ASC, au_fname DESC
```

NULLs are always grouped together (i.e., considered equal) for the purposes of sorting. Depending on your platform, NULLs will be clumped together at the top or at the bottom of the result set. The following query on SQL Server:

```
SELECT title, price
FROM titles
ORDER BY price, title
```

provides this result set (edited for brevity):

title	price
Net Etiquette	NULL
The Psychology of Computer Cooking	NULL
The Gourmet Microwave	2.9900
You Can Combat Computer Stress!	2.9900
Life Without Fear	7.0000
Onions, Leeks, and Garlic: Cooking Secrets of the Me	20.9500
Computer Phobic AND Non-Phobic Individuals: Behavior	21.5900
But Is It User Friendly?	22.9500

You can force NULLs to appear at the top or bottom of the result set using *ASC* or *DESC*. Of course, all the non-NULL rows of the result set are also ordered in ascending or descending order too.

The ANSI standard for *ORDER BY* also supports the use of columns for the *sort_expression* that are not referenced in the select item list. For example, the following query is valid under SQL2003:

```
SELECT title, price
FROM titles
ORDER BY title_id
```

Looking at the example above, you can see that although the query does not select **title_id**, column is the primary *sort_expression*. The result set is returned in **title_id** order even though that column is not selected.

Programming Tips and Gotchas

When using set operators (*UNION*, *EXCEPT*, *INTERSECT*), only the last query may have an *ORDER BY* clause. You should not use *ORDER BY* in subqueries of any type.

A number of behaviors that were supported in SQL92 are deprecated in SQL2003. You should avoid these usages:

References to table aliases

For example, *ORDER BY e.emp_id* should be changed to *ORDER BY emp_id*. If there is an ambiguous column name, then use an alias to compensate.

References to ordinal position

Use explicitly defined column aliases to compensate.

You may sort not only on columns, but also on expressions involving columns or even literals:

```
SELECT SUBSTRING(title,1,55) AS title, (price * 1.15) as price
FROM titles
WHERE price BETWEEN 2 and 19
ORDER BY price, title
```

When sorting on expressions from the select item list, you should use aliases to make the *ORDER BY sort_expression* column references easier!

DB2

DB2 supports the ANSI syntax, except for the *COLLATE* option. It also has the *ORDER OF* extension:

```
ORDER BY {ORDER OF table_name | sort_expression
        [{ASC | DESC}] }
```

where:

```
ORDER OF table_name
```

Tells DB2 to order the rows of a parent table as they are ordered in the subquery. To understand this concept more fully, remember that DB2 supports ordering at the subquery level. However, the ordered subquery does not return ordered result set unless the *ORDER BY ORDER OF* clause is used. The *table_name* must be the name of a table referenced in the parent query. For example:

```
SELECT qty, price, order_nbr
FROM
  (SELECT order_nbr, qty, price FROM sales_Q1_2005
   UNION
   SELECT order_nbr, qty, price FROM sales_Q2_2005
   ORDER BY price, qty, order_nbr) AS ytd_sales
ORDER BY ORDER OF ytd_sales
```

DB2 still supports SQL92 functions, such as ordering by ordinal position. By default, DB2 sorts NULL values as higher than all other values.

Another example of order subqueries follows. First, we will create a view called **best** that returns the top five best selling books from the **sales** table, and then we will find the top five authors in order:

```
CREATE VIEW best(title) AS
  SELECT title FROM sales
  ORDER BY sold
  FETCH FIRST 5 ROWS ONLY
SELECT author
FROM best JOIN authors ON best.title = author.title
ORDER BY ORDER OF best
```

DB2 *does not* support *ORDER BY* on columns of the following datatypes: *LONG VARCHAR*, *CLOB*, *LONG VARGRAPHIC*, *DBCLOB*, *BLOB*, and *DATALINK*. DB2 also does not support ordering by structure types.

MySQL

MySQL supports the ANSI standard, except for the *COLLATE* option.

You should not attempt to *ORDER BY* columns of the *BLOB* datatype because only the first bytes, defined by the *max_sort_length* setting, are used in the sort. By default, MySQL sorts NULL values as lowest (first) for *ASC* order and highest (last) for *DESC* order.

Oracle

Oracle supports the ANSI syntax, except for the *COLLATE* option. It also supports the *SIBLINGS* and *NULLS {FIRST | LAST}* options. Oracle's *ORDER BY* syntax is:

ORDER [SIBLINGS] BY

{sort_expression [ASC | DESC] [NULLS {FIRST | LAST}]} [...]

where the Oracle-specific keywords are:

ORDER [SIBLINGS] BY *sort_expression*

Sorts the result set of the query in order of the *sort_expressions*. A *sort_expression* may be a column name, an alias, an integer indicating a column's ordinal position, or another expression (e.g., *salary * 1.02*). The *ORDER SIBLINGS BY* clause tells Oracle to preserve any ordering specified by a hierarchical query clause (*CONNECT BY*), and to use the sort expression order for ordering of siblings in the hierarchy.

NULLS {FIRST | LAST}

NULLS FIRST and *NULLS LAST* specify that the records containing NULLs should appear either first or last, respectively. By default, Oracle places NULLs last for ascending order sorts, and first for descending order sorts.

You can emulate the behavior of the *COLLATE* option for a single session by using the *NLSSORT* function with the *NLS_SORT* parameter. You can explicitly emulate the behavior of the *COLLATE* option for all sessions on the server by using the *NLS_SORT* initialization parameter or, implicitly, with the *NLS_LANGUAGE* initialization parameter.

Oracle continues to support deprecated SQL92 features such as sorting by ordinal position. You should *not* perform an *ORDER BY* on any *LOB* column, nested table, or *VARRAY*.

PostgreSQL

PostgreSQL supports the ANSI SQL2003 standard, with the exception of the *COLLATE* option. It also supports the *USING* extension:

ORDER BY *{sort_expression1 [ASC | DESC | USING operator]}* [...]

where:

USING *operator*

Specifies a specific comparison operator. Thus, you may sort by *>*, *<*, *=*, *>=*, *<=*, and so forth. Ascending order is the same as specifying *USING <*, while descending order is the same as *USING >*.

PostgreSQL sorts NULLs as the highest values. Thus, NULLs will appear at the end when sorting in ascending order, and at the beginning when sorting in descending order.

SQL Server

SQL Server supports the ANSI standard including the *COLLATE* option. For example, the following query retrieves author firstnames from the **authors** table in the *SQL_Latin1* collation:

```
SELECT au_fname  
  
FROM authors  
  
ORDER BY au_fname  
  
COLLATE SQL_Latin1_general_cp1_ci_as
```

SQL Server continues to support the deprecated SQL92 features of the *ORDER BY* clause, including the ability to specify *ORDER BY* columns by their ordinal positions.

By default, SQL Server sorts NULL values higher than all other values.

You should not use *TEXT*, *IMAGE*, or *NTEXT* columns as *sort_expressions* on SQL Server.

See Also

SELECT

RELEASE SAVEPOINT Statement

The *RELEASE SAVEPOINT* statement eliminates one or more previously created savepoints in the current transaction.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
RELEASE SAVEPOINT savepoint_name
```

Keywords

savepoint_name

Represents a named savepoint (or target specification) created earlier in the transaction with the *SAVEPOINT* statement. The *savepoint_name* must be unique within the transaction.

Rules at a Glance

Use the *RELEASE SAVEPOINT* statement within a transaction to destroy a named savepoint. Any savepoints that were created after the named savepoint will also be destroyed.

To illustrate the behavior of savepoints, the following example code inserts a few records, creates a savepoint named **first_savepoint**, and then releases it:

```
INSERT authors (au_id, au_lname, au_fname, contract )  
  
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);  
  
SAVEPOINT first_savepoint;
```

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);

INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);

RELEASE SAVEPOINT first_savepoint;

COMMIT;
```

In this example, the **first_savepoint** savepoint is destroyed and then all three records are inserted into the **authors** table.

In the next example, we perform the same action but with more savepoints:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);

SAVEPOINT first_savepoint;

INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);

SAVEPOINT second_savepoint;

INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);

SAVEPOINT third_savepoint;

RELEASE SAVEPOINT second_savepoint;

COMMIT;
```

In the earlier example, when we release the savepoint called **second_savepoint**, the database actually releases **second_savepoint** and **third_savepoint**, since **third_savepoint** was issued after **second_savepoint**.

Once released, a savepoint name can be reused.

Programming Tips and Gotchas

Issuing either a *COMMIT* or full *ROLLBACK* statement will destroy all open savepoints in a transaction. Issuing a *ROLLBACK TO SAVEPOINT* statement returns the transaction to its state at the specified savepoint. Thus, any savepoints declared afterwards are nullified.

DB2

DB2 supports the *RELEASE SAVEPOINT* statement fully with one optional keyword:

```
RELEASE [TO] SAVEPOINT savepoint_name
```

where:

TO

An optional noise word.

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

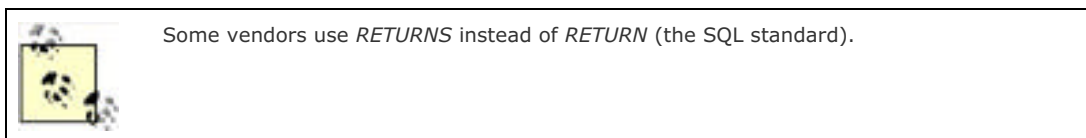
Not supported.

See Also

ROLLBACK
SAVEPOINT

RETURN Statement

The *RETURN* statement terminates processing within a SQL-invoked function (as opposed to a host-invoked function) or stored procedure and returns the routine's result value.



Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Supported, with limitations
SQL Server	Supported

SQL2003 Syntax

RETURN return_parameter_value | NULL

Keywords

return_parameter_value

Represents a value returned by the routine code, and can be any sort of a wide variety of values.

NULL

Terminates the function without returning an actual value

Rules at a Glance

Use the *RETURN* statement within procedural code to terminate processing. For example, you might create a user-defined function that takes a complex and often-used *CASE* expression and, when passed a parameter, returns a single, easy-to-understand expression value.

Programming Tips and Gotchas

Although the *RETURN* statement is categorized as a separate command within SQL, it is deeply intertwined with the *CREATE FUNCTION* and *CREATE PROCEDURE* statements. Consequently, the *RETURN* statement is almost always found embedded in one of these other commands. Check each statement's section in this book, or your vendor documentation, to get a more complete understanding of each platform's implementation of *RETURN* within the context of each statement.

DB2

DB2 uses the *RETURN* statement to exit a routine and, in a procedure, return any integer value. In a user-defined function, the *RETURN* statement returns the actual value that is calculated by the function. DB2 also allows query-based results to be returned by the statement, as in the following:

```
RETURN [ {expression | NULL | WITH table_name  
      ( column [...]) [AS (select_statement) ] } ] select_statement
```

where:

expression

Returns a literal or variable to the client or calling program.

NULL

Returns a null value of the datatype defined in the return clause for the calling user-defined function or method.

WITH *table_name* (*column* [...])

Acts as a *common table expression* (in other words, an ad hoc view) with each *column* keyword defining a column of the subsequent *select_statement*.

AS (*select_statement*) *select_statement*

Specifies the record or records to be returned from the function executing the *RETURN* statement. The number of columns in the record(s) retrieved by the second *select_statement* must match the number of columns in the *WITH* clause. This clause is not allowed in stored procedures.

If *RETURN* is used in a procedure, the value of *expression* must be an integer, but not in a user-defined function. The *NULL* and *WITH columns AS select_statement* clauses can be used with user-defined functions and methods, but not stored procedures. Here's a simple example of *RETURN*:

BEGIN

...<some SQL code>...

GOTO ERROR_HANDLER

... <some SQL code>...

SUCCESS: RETURN 0

ERROR_HANDLER: RETURN -200

END

If you need to check the *RETURN* value of a stored procedure that is called by another stored procedure, use the *GET DIAGNOSTICS* statement to find it. Refer to the DB2 documentation for more information on *GET DIAGNOSTICS*.

MySQL

Not supported. All procedural code in MySQL is written in C or C++. You can use C or C++ programming techniques to return value(s) to the client.

Oracle

Oracle supports the ANSI standard syntax for *RETURN*, excluding the *NULL* keyword. (Oracle does support the return of a *NULL* value, just not using the ANSI syntax.) Oracle allows the *RETURN* clause only in user-defined functions and user-defined operators. The returned value cannot be a *LONG*, *LONG RAW*, or *REF* datatype in the *CREATE OPERATOR* statement. PL/SQL user-defined functions fully support Boolean datatypes internally, but you cannot invoke a Boolean UDF from a SQL statement. Therefore, just use *INT* (0 or 1) or *VARCHAR2* ('TRUE' or 'FALSE') to hold Boolean values.

This example creates a function. The function returns the value that is stored in the **proj_rev** variable to the calling session:

```
CREATE FUNCTION project_revenue (project IN varchar2)
RETURN NUMBER
AS
    proj_rev NUMBER(10,2);
BEGIN
    SELECT SUM(DECODE(action,'COMPLETED',amount,0) -
        SUM(DECODE(action,'STARTED',amount,0) +
        SUM(DECODE(action,'PAYMENT',amount,0)
    INTO proj_rev
    FROM construction_actions
    WHERE project_name = project;
    RETURN (proj_rev);
END;
```

PostgreSQL

PostgreSQL supports the ANSI standard syntax for *RETURN*, excluding the *NULL* keyword:

```
RETURNS return_parameter_value | NULL
```

PostgreSQL allows you to define user-defined functions using either SQL code or C/C++. (PostgreSQL does not currently allow stored procedures, though you can duplicate their functionality with user-defined functions). For the purposes of this discussion, we are only interested in SQL functions.

The output allowed by the *return_parameter_value* may be a base type, complex type, *SETOF* type, *OPAQUE* modifier, or the same as the type of an existing column. The *SETOF type* modifier is used to return a set of items via the *RETURNS* statement, rather than a single data value. The *OPAQUE* modifier indicates the *RETURNS* statement does not return a value. *OPAQUE* can only be used within triggers.

SQL Server

SQL Server supports the *RETURN* statement with this syntax:

```
RETURN [return_integer_value]
```

The *RETURN* command is typically used in stored procedures or user-defined functions. The command causes an

immediate and complete exit from the program and, optionally, returns an integer value upon exit. SQL Server procedures implicitly return a zero value if no *RETURN* clause is supplied in the procedure definition. Any commands that follow the *RETURN* statement are ignored.

The *RETURN* statement in the following function returns an integer representing the calculated value:

```
CREATE FUNCTION metric_volume -  
  
    Input dimensions in centimeters.  
  
    (@length decimal(4,1),  
    @width decimal(4,1),  
    @height decimal(4,1) )  
  
RETURNS decimal(12,3) -- Cubic Centimeters.  
  
AS  
  
BEGIN  
  
    RETURN ( @length * @width * @height )  
  
END  
  
GO  
  
REVOKE
```

This example creates a function that returns a calculated value, the metric volume of a space, to the calling session.

See Also

CREATE/ALTER FUNCTION PROCEDURE
CREATE/ALTER TRIGGER

REVOKE Statement

The *REVOKE* statement takes two main forms. The first form of the statement removes specific statement permissions from a user, group, or role. The second form of the statement removes access permissions to specific database objects or resources.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 syntax for *REVOKE* takes this general form:

```
REVOKE { [special_options] | {privilege [...] | role [...]} }  
  
ON database_object_name  
  
FROM grantee_name [...]
```

[GRANTED BY {CURRENT_USER | CURRENT_ROLE}]

{CASCADE | RESTRICT}

Keywords

special_options

Allows the use of one of three optional *special_options*:

GRANT OPTION FOR

Undoes the *WITH GRANT OPTION* privilege assigned to a user, meaning that the user can no longer grant privileges to other users on the object but their own privileges on the object are still intact. (This clause is valid to revoke a privilege, but not to revoke a role.) Refer to the [GRANT Statement](#) section for more details.

HIERARCHY OPTION FOR

Undoes the *WITH HIERARCHY OPTION* privilege that allows a user to *SELECT* not only from the named table, but also all of its subtables. (This clause is valid to revoke a privilege, but not to revoke a role.)

ADMIN OPTION FOR

Undoes the ability to grant a role to other users. (This clause is valid to revoke a privilege, but not to revoke a role.)

privilege

Revokes privileges for a variety of statements, which may all be combined in any order:

ALL PRIVILEGES

Revokes all privileges currently assigned to the named users and/or for the specified database objects. Generally not a recommended approach, since it can encourage sloppy coding.

EXECUTE

Revokes the privileges to execute a routine (i.e., a stored procedure, user-defined function, or method).

{*SELECT* | *INSERT* | *UPDATE* | *DELETE*}

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked for *SELECT*, *INSERT*, and *UPDATE* privileges.

REFERENCES

Revokes the user's privilege to create any constraints or assertions referencing the database object as a parent object. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

TRIGGER

Revokes the user's ability to create triggers on the specified tables. As a side effect of the revoke, any triggers that depend on the privilege are also dropped.

UNDER

Revokes the privilege to create subtypes or typed tables.

USAGE

Revokes privileges to use a domain, user-defined type, character set, collation, or translation.

role

Revokes a specific, pre-defined role from the named grantee identified *FROM* clause. For example, the database administrator might want to create a role called **Reporter** that has read-only access to several tables. When the role is granted to the user, the user is able to do all of the actions the role may do. See the [GRANT Statement](#) section for more details.

ON *database_object_name*

Revokes privileges on a specific, predefined database object identified by the *database_object_name* provided. SQL2003 does not include support for system privileges but many implementations provide a variation of this clause to revoke such privileges. (The *ON* subclause is not used at all when revoking system privileges or roles.) The *database_object_name* may be one of the following:

```
{ [TABLE] object_name | DOMAIN object_name |  
  COLLATION object_name |  
  CHARACTER SET object_name | TRANSLATION object_name |  
  TYPE object_name / [SPECIFIC] {ROUTINE | FUNCTION |  
  PROCEDURE | METHOD}  
  object_name }
```

FROM *grantee_name*

Names the user(s) or role(s) who will lose the specified privilege. The *PUBLIC* keyword may also be used to revoke privileges granted to the *PUBLIC* global user list. Multiple grantees can be listed with a comma between each.

FROM { *CURRENT_USER* | *CURRENT_ROLE* }

Optionally used to specify who granted the privilege to the user in the first place. For example, when using *...FROM CURRENT_USER*, the privilege are revoked only if the current user granted the privilege. Otherwise, the statement would fail. When omitted, *CURRENT_USER* is the default.

RESTRICT | *CASCADE*

Limits a revoke operation to only the specified privilege (*RESTRICT*) or causes the specified privilege and all dependent privileges to be revoked (*CASCADE*), and possibly cause objects that depend on the privilege to be dropped. Note that *REVOKE...RESTRICT* will fail if dependent privileges exist. The dependent privileges must be revoked first.

Rules at a Glance

A specific privilege on a specific database object can be revoked for a single user using *REVOKE privilege_name ON object_name FROM grantee_name RESTRICT*. A specific privilege on a specific object may be revoked from all users via the *PUBLIC* global user list.

When revoking privileges from multiple grantees, simply place a comma between each. You may also revoke privileges from one or more grantees and the *PUBLIC* role in a single *REVOKE* statement. (*PUBLIC* is described in detail under the [GRANT Statement](#) section. When a grant is issued to *PUBLIC*, that means everyone has permissions to the object without the need for a grant specifically to the object.)

When revoking privileges on a table, the privileges may extend to the column level by showing a list of columns enclosed in parentheses after the table name.

Programming Tips and Gotchas

Most of the platforms isolate privileges at the user and role level. (Remember, a *role* is a group of privileges.) Thus, an individual user who is assigned two roles may have an individual set of permissions three times. In this situation, you would have to revoke the user's privileges directly and remove the user from both roles to completely negate their permissions.

An important aspect about *REVOKE* (and its sister command *GRANT*) is that certain elements of the command are geared toward object-level permissions, while other options of the command are more oriented toward roles or administrative privileges. Generally, these operations are never combined. The differences in object-level permissions and administrative privileges are fully explained under each database platform section below. For example, you might want to revoke certain object-level privileges for a given role (**salespeople**) or individual user (**e_fudd** and **prince_edward**):

```
REVOKE SELECT ON TABLE authors FROM salespeople RESTRICT;  
  
REVOKE ALL PRIVILEGES ON TABLE sales FROM e_fudd,  
  
prince_edward CASCADE;
```

A lot of table-level privileges can even be assigned down to the column level. Those privileges can be revoked:

```
REVOKE INSERT(au_id, au_fname, au_lname)  
  
ON authors FROM e_fudd;
```

The special option clauses (*GRANT OPTION*, *HIERARCHY OPTION*, and *ADMIN OPTION*) all exist to allow you to revoke the ability of users to pass privileges or roles on to other users. However, these clauses do not prevent the first group of users themselves from exercising those privileges. For example, we no longer want anyone in the role **manager** to grant their *UPDATE* privileges to other users:

```
REVOKE GRANT OPTION FOR UPDATE ON sales FROM manager CASCADE;
```

You can also remove role privileges from a user with the *REVOKE* command:

```
REVOKE manager FROM e_fudd CASCADE;
```

A common best practice is to keep *REVOKE* and *GRANT* statements as self-contained and logical as possible. Under this practice, you should avoid using the *CASCADE* and *ALL PRIVILEGES* clauses because they perform work for you that may not be immediately evident within the scope of the command.

DB2

DB2 offers a variety of additional options in its implementation of the *REVOKE* command:

```
REVOKE { [database_privileges [,...]] ON DATABASE } |  
  
[CONTROL ON INDEX index_name] |  
  
[ {BIND | CONTROL | EXECUTE} ON PACKAGE package_name] |  
  
[ {ALTERIN | CREATEIN | DROPIN} ON SCHEMA schema_name] |  
  
[PASSTHRU ON SERVER server_name] |  
  
[EXECUTE ON [SPECIFIC] {FUNCTION | PROCEDURE |  
  
METHOD} routine_name] |  
  
[object_privilege [,...]] ON [TABLE] object_name] |  
  
[{ALTER | USAGE} ON SEQUENCE sequence_name] |  
  
[USE OF TABLESPACE tablespace_name] } }  
  
[FROM [ {USER | GROUP} ] { grantee_name [,...]} |
```

group_name [,...] | PUBLIC }]

[RESTRICT]

where:

database_privileges

Revokes one or more privileges that apply to the entire database. The database privileges are:

BINDADD

Grants the authority to create packages.

CONNECT

Grants the authority to access the database. When *CONNECT* is revoked, all other explicitly granted privilege still exist, but are disabled until *CONNECT* privileges are regranted.

CREATETAB

Grants the authority to create tables.

CREATE_NOT_FENCED

Grants the authority to create functions that are not fenced into a specific user process; that is, they may execute within the database manager's process.

DBADM

Grants the superuser authority, meaning the ability to create any object, fully access any object created by other users, and grant those privileges to other users.

IMPLICIT_SCHEMA

Grants the authority to implicitly create a schema, though this privilege doesn't impact the ability to explicitly create a schema or any objects within a schema.

LOAD

Grants the authority to explicitly *LOAD* a database.

CONTROL ON INDEX *index_name*

Revokes the privilege to drop an index, once created, of the specified *index_name*. If you leave off the *FROM* clause, the current user context is assumed.

{ BIND | CONTROL | EXECUTE } ON PACKAGE *package_name*

Revokes the privileges associated with packages. Note that *RUN* and *PROGRAM* are acceptable synonyms for *EXECUTE* and *PACKAGE*, respectively. If you leave off the *FROM* clause, the current user is assumed if all privileges are user-level (GRANTEETYPE of U) or the current group is assumed if all the privileges are group-level (GRANTEETYPE of G). If the privileges are a mix of user- and group-level, an error will result. The package privileges are:

BIND

Grants the authority to bind or rebind the specified package. *CONTROL* privileges must also be revoked,

if the grantee has them, to properly remove *BIND* privileges.

CONTROL

Grants the authority to drop the specified package, as well as the authority to extend privileges to other grantees.

EXECUTE

Grants the authority to execute a package. Like *BIND*, *CONTROL* privileges must also be revoked, if the grantee has them, to properly remove *EXECUTE* privileges.

{ ALTERIN | CREATEIN | DROPIN } ON SCHEMA *schema_name*

Revokes the privileges associated with manipulating a schema (that is, a user-owned set of objects). Unlike some of the other optional clauses of this DB2 command, the *ON SCHEMA* syntax requires the *FROM* clause. Using this command does not supercede any specific privileges granted within a schema. Thus, you might revoke privileges on a user or group using the *ON SCHEMA* clause but the user or group might still be able to do any number of other activities within that schema because of other grant statements.

ALTERIN

Grants the authority to alter or comment on objects in a schema defined by *ON SCHEMA schema_name*.

CREATEIN

Grants the authority to create objects in a schema defined by *ON SCHEMA schema_name*.

DROPIN

Grants the authority to drop objects in the schema defined by *ON SCHEMA schema_name*.

PASSTHRU ON SERVER *server_name*

Revokes the privilege to access an external data source of *server_name* using pass-through mode.

EXECUTE ON

Revokes privileges on an ANSI *routine* (i.e., a user-defined function, user-defined method, or stored procedure). The elements of this clause are as follows:

[SPECIFIC] FUNCTION *function_name*

Specifies that the user loses execution privileges on a specific function. A *SPECIFIC* function name may be specified. DB2 also supports functions without names. In this case, *FUNCTION [(param [,...])]* should be used, including any and all parameters of the function. If *routine_name* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific function name to revoke from a user the execution privilege on all functions in the current or specified schema.

PROCEDURE [*schema.*]*procedure_name*

Specifies that the user loses execute privileges on all stored procedures in the specified schema, even those created in the future. If *schema* is not included, the current schema is assumed. You may substitute an asterisk (*) for a specific stored procedure name to revoke the execution privilege from a user on all stored procedures in the current or specified schema.

[SPECIFIC] METHOD *method_name*

Specifies that the user loses rights to execute all methods of *routine_name*, even those that have not yet been created. A *SPECIFIC* method name may be specified. DB2 also supports methods without names. In this case, *METHOD [(param [,...])]* can be used, including any and all parameters of the function. Alternately, the asterisk

(*) can be used to specify that all types in the schema are revoked from the user.

object_privilege

Revokes privileges for a variety of statements on tables, views, and nicknamed tables or views. These may be combined in any order:

ALL [PRIVILEGES]

Revokes all privileges currently assigned to the named users and/or for the specified database objects as a shorthand for every privilege the grantor has available to grant. The *PRIVILEGES* keyword is optional. We do not recommend this approach since it can encourage sloppy programming.

ALTER

Revokes the privilege to add or alter columns, constraints, comments, or triggers on a specified table or view and restricts the ability to change nicknamed column names or datatypes or to change a column option for a nickname.

CONTROL

Revokes the ability to drop the table or view or execute a RUNSTATS utility against it.

INDEX

Revokes the ability to create an index against a table or an index specification on a nickname.

{SELECT | INSERT | UPDATE | DELETE}

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked for the *SELECT*, *INSERT*, and *UPDATE* statements.

REFERENCES

Revokes the user's privilege to create or drop referencing constraints (such as *CHECK* or *FOREIGN KEY*) referencing the database object as a parent object.

*{ALTER | USAGE} ON SEQUENCE *sequence_name**

Revokes privileges on a sequence in an explicit schema or the current schema (*USAGE*), or the privilege to execute the *ALTER SEQUENCE* statement (*ALTER*).

*USE OF TABLESPACE *tablespace_name**

Revokes the *USE* privilege on a tablespace. Note that the user might be able to continue to make tables in the specified tablespace if rights on the tablespace are granted to *PUBLIC*, or if the user is in a group (such as *DBADM*) that also has privileges in the specified tablespace.

*FROM { [USER | GROUP]} *grantee_name**

Names the user or users who will lose the specified privilege. When both a user and group exist with the same name, use the optional *GROUP* or *USER* keywords to control where the privileges will be dropped. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role. Multiple grantees can be listed with a comma between each.

RESTRICT

Identical to the ANSI usage, except that *RESTRICT* is required on *SCHEMA* and *FUNCTION* privileges.

DB2 supports a number of privileges, as described above, through a specific user. In the following example, the user **emily** loses the create table (*CREATETAB*) privilege:

```
REVOKE CREATETAB ON DATABASE FROM emily
```

This revokes the privilege of creating tables from user **emily**.

The following statement revokes the privilege of creating packages from the group **student**. The *FROM GROUP* clause should always be used when a user and group exist with the same name.

```
REVOKE BINDADD ON DATABASE FROM GROUP student
```

The first command in the next example revokes the privilege to drop the index **upkcl_aidind** from the user **dylan** and the group of users called **editors**, while the second command revokes the tablespace privileges of the user **sam**.

```
REVOKE CONTROL ON INDEX upkcl_aidind
```

```
FROM USER dylan, GROUP editors
```

```
REVOKE USE OF TABLESPACE futures FROM USER sam
```

The next example revokes the ability to execute the package **publications** by both the group **editors** and the user **jake**.

```
REVOKE EXECUTE
```

```
ON PACKAGE publications
```

```
FROM GROUP editors, USER jake
```

```
REVOKE CREATEIN ON SCHEMA pubs FROM anna
```

The next example revokes the ability of user **anna** to create new objects in the schema **pubs**, while the second command revokes the privilege to pass through to data source **east**.

```
REVOKE PASSTHRU ON SERVER east FROM USER anna
```

Specific object-level privileges are also easy to grant and revoke. In some cases, you can issue the same command in several ways, depending upon whether you include a lot of the optional keywords. For example, the following two commands do the same thing:

```
REVOKE UPDATE ON employee FROM sam
```

```
REVOKE UPDATE ON TABLE employee FROM USER sam
```

In both of the examples above, the user **sam** loses the privilege to update the **employee** table. You can also combine object-level privileges and users and/or groups in a single command. For example, the following command revokes all the write privileges (*INSERT*, *UPDATE*, *DELETE*) from the users **katie** and **jake**, as well as from the group **editors**:

```
REVOKE INSERT, UPDATE, DELETE ON employee FROM USER katie,
```

```
USER jake, GROUP editors
```

DB2 ships with the *PUBLIC* group, which includes all users. In addition, there are several levels of system privileges on a DB2 installation whose privileges cannot be modified:

SYSADM

Issues system administration privileges (the highest level of authority). Users with this level have all privileges in all databases.

DBADM

Issues database-wide privileges. Users with this level have all privileges in a specific database, though they cannot execute operations that require administrative access to system resources (such as the DB2 statement *CREATE TABLESPACE*).

SYSCtrl

Issues system control privileges without data access privileges. Users with this level have the ability to create, modify, and drop databases and tablespaces, and are able to halt instances or databases.

SYSMaint

Issues system control privileges without data access privileges (at a lower level than *SYSCtrl*). Users with this level have the ability to update database configuration files, issue backups and restores, and monitor a database.

Users can be granted and revoked these system-level privileges, but the privileges of the specific scope of each system-level privilege (system-wide, database-wide, etc.) cannot be altered.

MySQL

The *REVOKE* statement rolls back permissions previously granted to one or more users. MySQL supports many of the standard keywords of the ANSI standard. Notable exceptions for support include *TRIGGER*, *EXECUTE*, and *UNDER*. MySQL also has a nice shortcut to allow global assignment or revocation of privileges. Please refer back to the ANSI standard description for those elements of the command not described below:

```
REVOKE [ { ALL [PRIVILEGES] |
{SELECT | INSERT | UPDATE} [ (column_name [,...]) ] | DELETE |
REFERENCES [ (column_name [,...]) ] } |
[ USAGE ] | [{ALTER | CREATE | DROP}] | [FILE] | [INDEX] | [PROCESS] |
[RELOAD] | [SHUTDOWN] | [CREATE TEMPORARY TABLES] | [LOCK TABLES] |
[REPLICATION CLIENT] | [REPLICATION SLAVE] | [SHOW DATABASES] |
[SUPER] },...]
ON {table_name | * | *.* | database_name.*}
FROM user_name [,...]
```

where:

ALL [PRIVILEGES]

Synonymous with *ALL PRIVILEGES*. On MySQL, this includes all privileges that apply at the level indicated by the *ON* clause, except the *GRANT OPTION*. (This includes privileges like *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and others.)

SELECT | INSERT | UPDATE | DELETE

Revokes the ability to read, write, modify, and erase data from tables (and possibly from specific columns of a table), respectively.

REFERENCES

Unimplemented.

USAGE

Revokes all of a user's privileges.

{ALTER | CREATE | DROP} ON *table_name*

Revokes the ability to alter, create, or drop tables and other database objects.

FILE

Revokes the ability to load data from or write data to files using the *SELECT INTO* and *LOAD DATA* commands.

INDEX

Revokes the ability to create or drop indexes.

PROCESS

Revokes the ability to view running processes using *SHOW FULL PROCESSLIST*.

RELOAD

Revokes the ability to invoke the *FLUSH* command.

SHUTDOWN

Revokes usage of the *mysqladmin shutdown* command to kill the server process.

CREATE TEMPORARY TABLES

Revokes the ability to create temporary tables.

LOCK TABLES

Revokes the use of the MySQL command *LOCK TABLES* on tables where the user has *SELECT* privileges.

REPLICATION CLIENT

Revokes the privilege to see metadata about replication slaves and masters.

REPLICATION SLAVE

Revokes the right to read binlogs of a replication master from a replication slave.

SHOW DATABASES

Revokes the privilege to execute the MySQL command *SHOW DATABASES*.

SUPER

Revokes the user privilege of opening one connection even if the *max_connections* threshold is reached. Also revokes from the users the privileges to execute these MySQL commands: *CHANGE MASTER*, *KILL*, *mysqladmin debug*, *PURGE [MASTER] LOGS*, and *SET GLOBAL*.

ON {*table_name* | * | *.* | *database_name*.*}

Removes privileges from the specified *table_name*, from all tables within the current database with *, from all tables in the specified *database_name* with *database_name*.*, or from all tables in all databases with *.*.

FROM

Removes the privileges of one or more users, separated by commas. Usernames may also include an *@host_name* suffix, if you wish to restrict the revocation to a specific host computer.

REVOKE has size limitations. Usernames cannot be longer than 16 characters, while host, database, and database object names can be up to 60 characters in size. Usernames may also be linked to a specific host. Refer to the *GRANT* command for more details.



MySQL's implementation of *REVOKE* does not explicitly roll back permissions on objects that are dropped. Thus, it is necessary to explicitly *REVOKE* permissions on a table, even if the table is dropped. For example, let's say you drop a table without issuing a *REVOKE* on the existing permissions. If you later recreate the table, all of the old permissions will still be in place. Similarly, a user's privileges still persist in the database, even after the user has been dropped.

It is also important to note that MySQL allows multiple levels of permissions. Thus, a user might have access to a table from a table-level grant and have an additional set of permissions to the same table because they have been given global database or server-wide permissions. This means you need to be careful when revoking permissions because global level permissions might continue to provide permissions to a user when you thought you had revoked them!

The first command shown below revokes all privileges on the **sales** table for users **emily** and **dylan**, while the second command revokes select privileges for the user **kelly** in the current database:

```
REVOKE ALL PRIVILEGES ON sales FROM emily, dylan;
```

```
REVOKE SELECT ON * FROM kelly;
```

The first command below removes **kelly**'s ability to grant privileges to other users on the **sales** table, while the second command removes privileges that the user **sam** has in the **pubs** database:

```
REVOKE GRANT OPTION ON sales FROM kelly;
```

```
REVOKE ALL ON pubs.* FROM sam;
```

Oracle

The *REVOKE* command not only immediately revokes object and system privileges, it also revokes a role from a user (or a role from another role). Refer to the *GRANT* statement for more information on the specific object and system privileges supported by the *REVOKE* command.



The two forms of the *REVOKE* command, *REVOKE object_privilege* and *REVOKE system_privilege*, are mutually exclusive. Do not attempt to do both operations in a single statement. Since the complete syntax for both forms of the command are extremely long, refer to the [GRANT Statement](#) section for more details on the different *object_privileges* and *system_privileges*.

Oracle provides an interesting twist on privileges. While other database platforms typically allow a user to have more than one context for a set of privileges (based on the individual user and again for any groups that they are a member of), Oracle takes this one step further. A user may also have many grantors who have given them permission on a specific object. When this is the case, *ALL* grantors must revoke the privilege to effectively rid the user of the privilege. If even one grantor doesn't do this, then the user still has the privilege. Use the following syntax with *REVOKE*:

```
REVOKE { [object_privilege] [,...]  
        [system_privilege] |  
        [role] }  
[ON { [schema_name.][object] |  
      [DIRECTORY directory_object_name] |  
      [JAVA { [SOURCE | RESOURCE] } ] [schema_name.][object] } ]  
FROM {grantee_name [,...] | role_name [,...] | PUBLIC}
```

[CASCADE [CONSTRAINTS]] [FORCE];

where:

object_privilege

Revokes the specified privileges for a given user, *grantee_name*, or role against the named schema object, for example, a table or view.

ALL [PRIVILEGES]

Revokes all granted privileges on a named schema object. Since *ALL* also includes the *REFERENCES* privilege, you must include the *CASCADE* clause (see *REFERENCES* below).

ALTER

Revokes the privilege to change an existing table using the *ALTER TABLE* statement.

EXECUTE

Revokes the privilege to execute a stored procedure, user-defined function, or package.

INDEX

Revokes the privilege to create indexes on a table.

REFERENCES

Revokes the privilege to define referential integrity constraints. Requires that the *CASCADE CONSTRAINTS* clause also be used.

{ *SELECT* | *INSERT* | *DELETE* | *UPDATE* }

Revokes the privilege to execute each of the respective SQL commands against the named schema object. Note that *DELETE* privileges are dependent on *SELECT* privileges.

system_privilege

Revokes the specified Oracle system privilege from a given user, *grantee_name*, or role against the named system privilege such as *CREATE TRIGGER* or *ALTER USER*. Do not use the *ON* clause with the *REVOKE system_privileges* variation of the statement. Because there are so many and varied system privileges, please refer to the section under Oracle's implementation in the [GRANT Statement](#) section for a full list of system privileges.

role

Revokes a user from the named role or one role from another role.

ON

Revokes the named privilege for a given user or role from the named object. Objects include tables, views, sequences, stored procedures, user-defined functions, packages, materialized views, user-defined types, libraries, indextypes, user-defined operators, or a synonym for any of these. If you do not include the schema name, Oracle assumes the schema of the current user. For example, you could revoke *SELECT* privileges for a given user on the **scott.authors** table. Oracle also supports two additional keywords for special cases:

DIRECTORY *directory_object_name*

Identifies a directory object from which the privileges are revoked.

```
JAVA [ { SOURCE | RESOURCE } ] [schema_name.][object ] }
```

Identifies a Java source or a resource schema object from which the privileges are revoked.

```
FROM {grantee_name | role_name | PUBLIC}
```

Names the user or role who will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role. Multiple grantees can be listed with a comma between each.

```
CASCADE [CONSTRAINTS]
```

Drops any referential integrity constraints that depend on the privilege being revoked in the first place. This clause is needed only when the *REFERENCES* or *ALL [PRIVILEGES]* clauses are used.

```
[FORCE]
```

Required to revoke *EXECUTE* object privileges on user-defined type objects with type or table dependencies.

Oracle automatically cascades the revocation from the defined *grantee_name* to all users that received their privileges from the grantee. In addition, any objects created by the grantee that depend on the privilege (such as stored procedures, triggers, views, and packages that depend on a *SELECT* privilege against a certain table) will become invalid.

Users who are granted the *GRANT ANY ROLE* system privilege may revoke any role. The *REVOKE* command can only revoke privileges specifically granted with the *GRANT* command, not privileges available through roles or the operating system. In a case like this, you must use the *REVOKE* command to drop the privilege from the role. All users assigned to the role will then lose the privilege.

The following is an example of revoking a role from specific grantee and revoking a system privilege from a role:

```
REVOKE read-only FROM sarah;  
  
REVOKE CREATE ANY SEQUENCE,  
CREATE ANY DIRECTORY FROM read_only;
```

Here's an example that revokes a *REFERENCES* privilege and cascades the revoked privileges:

```
REVOKE REFERENCES  
ON pubs_new_york.emp  
FROM dylan  
CASCADE CONSTRAINTS;
```

The following example grants all privileges on a specific table, and then revokes a privilege:

```
GRANT ALL PRIVILEGES ON emp TO dylan;  
  
REVOKE DELETE, UPDATE ON emp FROM dylan;
```

PostgreSQL

PostgreSQL offers base-level features of the *REVOKE* statement, primarily for revoking privileges on specific tables, views, and sequences. It does not offer the ANSI standard special options like *HIERARCHY OPTION FOR* or *ADMIN OPTION FOR*. The syntax is as follows:

```
REVOKE { [ GRANT OPTION FOR ] |  
privileges [...] }
```

```
ON {object_name} [,...]  
FROM {grantee_name [,...]} | PUBLIC | GROUP group_name}  
[ {CASCADE | RESTRICT} ]
```

where:

GRANT OPTION FOR

Revokes the privilege of a user to grant privileges to other users.

privileges

Revokes privileges for a variety of statements, which may be combined in any order:

ALL PRIVILEGES

A shorthand for every privilege the grantor has available to grant. Revokes all privileges currently assigned to the named users and/or for the specified database objects. Generally not a recommended approach, since it can encourage sloppy coding.

```
{SELECT | INSERT | DELETE | UPDATE}
```

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

REFERENCES

Revokes the user's privilege to create or drop foreign key constraints that reference the database object as a parent object.

RULE

Revokes the user's privilege to create or drop a rule on a table or view.

USAGE

Revokes privileges to use a domain, user-defined type, or character set, as opposed to the *UNDER* privilege, which controls the ability to create these objects.

```
FROM grantee_name [,...]} | PUBLIC | GROUP group_name
```

Names the user or role who will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role (a role that implicitly includes all users). Multiple grantees can be listed with a comma between each.

CASCADE | RESTRICT

Limits a revoke operation to only the specified privilege (*RESTRICT*) or causes the specified privilege and all dependent privileges to be revoked (*CASCADE*). Only used with the *GRANT OPTION FOR* clause.

PostgreSQL's implementation of *REVOKE* is relatively straightforward. The only issue to be aware of is that PostgreSQL treats the term *GROUP* as a synonym for *ROLE*. For example, the following removes some privileges from the *PUBLIC* group and from the *READ-ONLY* group:

```
REVOKE ALL PRIVILEGES ON employee FROM public;
```

```
REVOKE SELECT ON jobs FROM read-only;
```

PostgreSQL *does not* support privileges on individual columns within a table or view. In order to work around this limitation, you can create a view with only the columns in question and *GRANT* or *REVOKE* privileges to and from that view.

When revoking the *GRANT OPTION FOR* clause, you should pay extra attention to dependencies. If you revoke a user's privilege with the *RESTRICT* keyword, the statement will fail if other users depend on the revokee. If you revoke a user's privilege with the *CASCADE* keyword, the statement will revoke privileges not only for the revokee, but for all users that depend on the revokee.

SQL Server

SQL Server implements the *REVOKE* statement as a means to undo any permission settings for a given user. This is significant because SQL Server supports an extension statement called *DENY* which explicitly disallows a user from a specific resource. *REVOKE*, in SQL Server, can be used to undo permissions granted to a user with *GRANT*. If you want to explicitly prevent a user from having a certain privilege, you must use the *DENY* statement.

SQL Server does not support the *HIERARCHY OPTION* and *ADMIN OPTION* clauses of the ANSI standard. Although the *ADMIN OPTION* clause is not supported, a couple of administrative privileges (*CREATE* and *BACKUP*) are supported in SQL Server's version of *REVOKE*. Use the following syntax:

REVOKE [GRANT OPTION FOR]

{ [object_privilege] [,...]

[system_privilege] }

[ON [object] [(column [,...])]]]

{TO | FROM} {grantee_name [,...] | role [,...] | PUBLIC | GUEST }

[CASCADE]

[AS {group_name | role_name}]

where:

GRANT OPTION FOR

Revokes the privilege of a user to grant a specific privilege to other users.

object_privilege

Revokes privileges for a variety of statements, which may be combined in any order:

ALL PRIVILEGES

Revokes all privileges currently assigned to the named users and/or for the specified database objects. We do not recommend this approach, since it can encourage sloppy programming.

{SELECT | INSERT | DELETE | UPDATE}

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

REFERENCES

Revokes the user's privilege to create or drop foreign key constraints that reference the database object as a parent object.

RULE

Revokes the user's privilege to create or drop a rule on a table or view.

EXECUTE

Revokes privileges to execute a stored procedure, user-defined function, or extended stored procedure.

system_privilege

Revokes the privilege to execute any of the following statements: *CREATE DATABASE*, *CREATE DEFAULT*, *CREATE FUNCTION*, *CREATE PROCEDURE*, *CREATE RULE*, *CREATE TABLE*, *CREATE VIEW*, *BACKUP DATABASE*, and *BACKUP LOG*.

ON *object* [(*column* [...])]

Removes the privilege for the user on the named object. If the object is a table or view, you may optionally revoke privileges on specific columns. You may revoke *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *REFERENCES* privileges on a table or view. You may revoke only *SELECT* and *UPDATE* privileges on the columns of a table or view. You may revoke *EXECUTE* privileges on a stored procedure, user-defined function, or extended stored procedure.

{TO | FROM} *grantee_name* | *role* | PUBLIC | GUEST

Names the user or role who will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role (a role that implicitly includes all users). Multiple grantees can be listed with a comma between each. SQL Server also supports the role known as *GUEST*, an account used by all users who do not have a user account in the database.

CASCADE

Revokes privileges on users who received their permissions under the *WITH GRANT OPTION* clause. Required when using the *GRANT OPTION FOR* clause.

AS {*group_name* | *role_name*}

Defines the authority by which the privilege is revoked. In certain circumstances, a user may need to temporarily assume the privileges of a group to revoke the specified privileges. To accommodate this scenario, you can use the *AS* clause to assume the authority of a given group.

The two forms of the *REVOKE* command, *REVOKE object_privilege* and *REVOKE system_privilege*, are mutually exclusive. Do not attempt to do both operations in a single statement. The key syntactical difference between them is that you should not include the *ON* clause when revoking system privileges. For example, to drop a system privilege:

```
REVOKE CREATE DATABASE, BACKUP DATABASE FROM dylan, katie
```

If commands were granted to a user *WITH GRANT OPTION* enabled, the privilege should be revoked using both *WITH GRANT OPTION* and *CASCADE* together. For example:

```
REVOKE GRANT OPTION FOR
```

```
SELECT, INSERT, UPDATE, DELETE ON titles
```

```
TO editors
```

```
CASCADE
```

```
GO
```

REVOKE can only be used in the current database. Consequently, the *CURRENT_USER* and *CURRENT_ROLE* options in the ANSI variations are always implied. *REVOKE* also is used to disable any *DENY* settings.



SQL Server additionally supports the *DENY* statement. *DENY* is syntactically the same as *REVOKE*. However, it is conceptually different in that *REVOKE* neutralizes a user's privileges, while *DENY* explicitly prohibits a user's privileges. Use the *DENY* statement to keep a user or role from accessing a privilege, even a privilege granted through a role or an explicit permission.

You must use *REVOKE* to remove previously granted *OR* denied privileges. For example, the user **kelly** took an extended leave of absence for maternity leave. During that time, her permissions on the employee table where denied. Now that she's returned, we'll lift the denied privileges:

```
DENY ALL ON employee TO Kelly
```

```
GO
```

```
REVOKE ALL ON employee TO Kelly
```

```
GO
```

In this example, the *REVOKE* command did not remove her privileges; rather, it neutralized the *DENY* statement.

See Also

GRANT

ROLLBACK Statement

The *ROLLBACK* statement undoes a transaction to its beginning or to a previously declared *SAVEPOINT*. *ROLLBACK* also closes any open cursors.

Platform	Command
DB2	Supported
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with variations

SQL2003 Syntax

```
ROLLBACK [WORK]
```

```
[AND [NO] CHAIN]
```

```
[TO SAVEPOINT savepoint_name]
```

Keywords

WORK

An optional keyword, but basically just noise.

AND [NO] CHAIN

AND CHAIN tells the DBMS to end the current transaction, but to share the common transaction environment (such as transaction isolation level) with the next transaction. *AND NO CHAIN* simply ends the transaction (and is effectively the same as not including the clause at all).

TO SAVEPOINT *savepoint_name*

Allows the transaction to be rolled back to a named savepoint (that is, a partial rollback) rather than rolling back the entire transaction. The *savepoint_name* may be a literal expression or a variable. If no savepoint of *savepoint_name* is active, the statement will return an error. When the *TO SAVEPOINT* clause is omitted, all cursors are closed. When the *TO SAVEPOINT* clause is included, only the cursors that were open within the savepoint are closed.

In addition to undoing a single data-manipulation operation such as an *INSERT*, *UPDATE*, or *DELETE* statement (or a batch of them), the *ROLLBACK* statement undoes transactions up to the last issued *START TRANSACTION*, *SET TRANSACTION*, or *SAVEPOINT* statement.

Rules at a Glance

ROLLBACK is used to undo a transaction. It can be used to undo explicitly declared transactions that are started with a *START TRAN* statement or implicitly with a transaction-initiating statement. It can also be used to undo implicit transactions that are started without a *START TRAN* statement. *ROLLBACK* is mutually exclusive of the *COMMIT* statement.

Most people associate commands like *INSERT*, *UPDATE*, and *DELETE* with the term "transaction." However, transactions encompass a wide variety of commands. This list of commands varies from platform to platform, but generally includes any command that alters data or database structures and is logged to the database logging mechanism. According to the ANSI standard, all SQL statements can be undone with *ROLLBACK*.

Programming Tips and Gotchas

The most important gotcha to consider is that some database platforms perform automatic and *implicit* transactions, while others require *explicit* transactions. If you assume a platform uses one method of transactions over the other, you may get bitten. Thus, when moving between database platforms, you should follow a standard, preset way of addressing transactions. We recommend staying with explicit transactions using *SET TRAN* or *START TRAN* to begin a transaction, and *COMMIT* or *ROLLBACK* to end a transaction.

DB2

DB2 supports the basic ANSI form of the statement. It does not support the *[AND [NO] CHAIN]* clause. (Also note that DB2 does not support the *START TRANSACTION* statement, so all transactions in DB2 start implicitly.) DB2's *ROLLBACK* syntax is as follows:

```
ROLLBACK [WORK] [TO SAVEPOINT savepoint_name]
```

A number of DB2 *SET* statements are not covered by transaction control and, thus, cannot be rolled back. These include: *SET CONNECTION*, *SET CURRENT DEFAULT TRANSFORM GROUP*, *SET CURRENT DEGREE*, *SET CURRENT EXPLAIN MODE*, *SET CURRENT EXPLAIN SNAPSHOT*, *SET CURRENT PACKAGESET*, *SET CURRENT QUERY OPTIMIZATION*, *SET CURRENT REFRESH AGE*, *SET EVENT MONITOR STATE*, *SET PASSTHRU*, *SET PATH*, *SET SCHEMA*, and *SET SERVER OPTION*.

DB2 issues an implicit rollback any time a unit of work terminates abnormally.

When a regular (non-savepoint) rollback occurs on DB2, all locks held by the unit of work are released, open cursors are closed, *LOB* locators are freed, and in some cases, caching is impacted. When a rollback to savepoint occurs on DB2, locks and *LOB* locators are preserved, and dynamically prepared SQL is still valid (though it will have to be parsed again). Rollback to savepoint involving cursors is a bit more complicated. If a cursor depends on DDL in the savepoint, it will be marked invalid. If a cursor is referenced in the savepoint but does not contain DDL used by the savepoint, the cursor will remain open and the cursor pointer will be positioned before the next row. In all other cases, the cursor remains unaffected.

MySQL

MySQL supports only the simplest rollback mechanism and, even then, you must declare any table you might issue a *ROLLBACK* against to be *transaction safe*. (A transaction safe table is one declared with the InnoDB or BDB property. Refer to *CREATE TABLE* for more information.)

```
ROLLBACK [TO SAVEPOINT savepoint_name]
```

MySQL allows you to issue transaction control statements like *COMMIT* or *ROLLBACK* against non-transaction safe tables, but it will simply ignore them and autocommit as usual. In the case of *ROLLBACK* against a non-transaction safe table, changes will not be rolled back.

MySQL, by default, runs in *AUTO COMMIT* mode, causing all data modifications to automatically be written to disk. You can set *AUTO COMMIT* off by issuing the command *SET AUTO COMMIT=0*. You can also control the autocommit behavior on a statement-by-statement basis using the *BEGIN* or *BEGIN WORK* command:

```
BEGIN;  
  
SELECT @A:=SUM(salary) FROM employee WHERE job_type=1;  
  
BEGIN WORK;  
  
UPDATE jobs SET summary=@A WHERE job_type=1;  
  
COMMIT;
```

MySQL automatically issues an implicit *COMMIT* upon the completion of any of these statements: *ALTER TABLE*, *BEGIN*, *CREATE INDEX*, *DROP DATABASE*, *DROP TABLE*, *RENAME TABLE*, and *TRUNCATE*.

MySQL supports partial rollbacks using savepoints with Version 4.0.14.

Oracle

Oracle supports the ANSI standard form of the *ROLLBACK* statement with the addition of the *FORCE* clause:

```
ROLLBACK [WORK] {[TO [SAVEPOINT] savepoint_name] |  
  
    [FORCE 'text']};
```

ROLLBACK clears all data modifications made to the current open transaction (or to a specific, existing savepoint). It also releases all locks held by the transaction, erases all savepoints, undoes all the changes of the current transaction, and ends the current transaction.

ROLLBACK ... TO SAVEPOINT rolls back just that portion of the transaction after the savepoint, erases all savepoints that followed, and releases all table- and row-level locks taken after the savepoint. Refer to the [SAVEPOINT Statement](#) section for more information.

Oracle's implementation closely follows the SQL standard with the exception of the *FORCE* option. *ROLLBACK FORCE* rolls back an in-doubt, distributed transaction. You must have the *FORCE TRANSACTION* privilege to issue a *ROLLBACK...FORCE* statement. *FORCE* cannot be used with *TO [SAVEPOINT]*. *ROLLBACK...FORCE* affects the transaction named in *'text'* and not the current transaction, where *'text'* must be equal to the local or global transaction ID of the transaction you want to rollback. (These transactions and their ID names are detailed in the Oracle system view **DBA_2PC_PENDING**.)

For example, you might want to roll back your current transaction to the **salary_adjustment** savepoint. These two commands are equivalent:

```
ROLLBACK WORK TO SAVEPOINT salary_adjustment;  
  
ROLLBACK TO salary_adjustment;
```

In the following example, you roll back an in-doubt distributed transaction:

```
ROLLBACK FORCE '45.52.67'
```

PostgreSQL

PostgreSQL supports the basic form of *ROLLBACK*, but it does not support savepoints.

```
{ROLLBACK | ABORT} [WORK | TRANSACTION];
```

ROLLBACK clears all data modifications made to the current open transaction. It will return an error if no transaction is currently open. PostgreSQL supports both the *WORK* option and the *TRANSACTION* option. It does not support rolling back to a savepoint. PostgreSQL supports *ABORT* as a synonym of *ROLLBACK*.

SQL Server

SQL Server supports both the *WORK* and *TRAN* keywords. The only difference between them is that the *ROLLBACK WORK* statement doesn't allow rolling back of a named transaction or to a specific savepoint.

```
ROLLBACK { [WORK] | [TRAN[SACTION]  
    [transaction_name | savepoint_name] ] }
```

If *ROLLBACK* is issued alone without the *WORK* or *TRAN* keywords, it rolls back all current open transactions. *ROLLBACK* normally frees locks, but it does not free locks when rolling back to a savepoint.

SQL Server allows you to name a specific *transaction_name* in addition to a specific *savepoint_name*. You may reference them explicitly, or you may use variables within Transact-SQL.

ROLLBACK TRANSACTION, when issued in a trigger, undoes all data modifications, including those performed by the trigger, up to the point of the *ROLLBACK* statement. Nested triggers are not executed if they follow a *ROLLBACK* within a trigger; however, any statements within the trigger that follow the rollback are not impacted by the rollback. *ROLLBACK* behaves similarly to *COMMIT* with regard to nesting, resetting the *@@TRANCOUNT* system variable to zero. (Refer to the [COMMIT Statement](#) section for more information on transaction control within a SQL Server nested trigger.)

Following is a Transact-SQL batch using *COMMIT* and *ROLLBACK* in Microsoft SQL Server. In the example, the code inserts a record into the **sales** table. If the insertion fails, the transaction is rolled back; if the insertion succeeds, the transaction is committed:

```
BEGIN TRAN -- initializes a transaction  
  
-- the transaction itself  
  
INSERT INTO sales  
  
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832')  
  
-- some error-handling in the event of a failure  
  
IF @@ERROR <> 0  
  
BEGIN  
    -- raises an error in the event log and skips to the end  
    RAISERROR 50000 'Insert of sales record failed'  
  
    ROLLBACK WORK  
  
    GOTO end_of_batch  
  
END  
  
-- the transaction is committed if no errors are detected  
  
COMMIT TRAN  
  
-- the GOTO label that enables the batch to skip to  
-- the end without committing  
  
end_of_batch:  
  
GO  
  
SAVEPOINT sales1
```

See Also

```
COMMIT  
RELEASE SAVEPOINT  
SAVEPOINT
```

SAVEPOINT Statement

This command breaks a transaction into logical breakpoints. Multiple savepoints may be specified within a single transaction. The main benefit of the *SAVEPOINT* command is that transactions may be partially rolled back to a savepoint marker using the *ROLLBACK* command.

Platform	Command
DB2	Supported, with variations
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

SAVEPOINT savepoint_name

Keywords

SAVEPOINT savepoint_name

Establishes a savepoint named *savepoint_name* within the current transaction.

Some vendors allow duplicate savepoint names within a transaction, but this is not recommended by the ANSI standard.

SQL2003 supports the statement *RELEASE SAVEPOINT savepoint_name*, enabling an existing savepoint to be eliminated. Refer to the [RELEASE SAVEPOINT Statement](#) section for more information about eliminating an existing savepoint.

Rules at a Glance

Savepoints are established within the scope of the entire transaction in which they are defined. Savepoint names should be unique within their scope. Furthermore, make sure you use *BEGIN* and *COMMIT* statements prudently, because accidentally placing a *BEGIN* statement too early or a *COMMIT* statement too late can have a dramatic impact on the way transactions are written to the database. Always make sure to provide easy-to-understand names for your savepoints because you'll be referencing them later in your programs.

Programming Tips and Gotchas

Generally, reusing a savepoint name won't produce an error or warning. A duplicate savepoint name will render the previous savepoint with the same name useless. So be careful when naming savepoints!

When a transaction is initiated, resources (namely, locks) are expended to ensure transactional consistency. Make sure that your transaction runs to completion as quickly as possible so that the locks are released for others to use.

DB2

DB2 supports a more elaborate version of *SAVEPOINT* than is specified in the ANSI standard.

SAVEPOINT savepoint_name [UNIQUE]

ON ROLLBACK RETAIN CURSORS [ON ROLLBACK RETAIN LOCKS]

where:

SAVEPOINT savepoint_name

Establishes a savepoint named *savepoint_name* within the current transaction.

UNIQUE

Indicates that the savepoint is intended to be unique and that the savepoint name cannot be reused or overwritten. (Supported only on DB2 for OS/390.)

ON ROLLBACK RETAIN CURSORS

Specifies that open cursors remain open when rolling back to the savepoint. (More details are given in the [ROLLBACK Statement](#) section.)

ON ROLLBACK RETAIN LOCKS

Specifies that locks are retained when rolling back to the named savepoint. This behavior is the default when the clause is omitted. (More details are given in the [ROLLBACK Statement](#) section.)

An important note about DB2 is that savepoints cannot be nested. An error will occur any time you issue a savepoint where an active savepoint already exists.

DB2 implicitly releases any savepoints within the transaction if it is committed.

In the following example, we set the **sales_processing** savepoint without explicitly retaining locks:

```
SAVEPOINT sales_processing ON ROLLBACK RETAIN CURSORS;
```

However, since retaining locks is the default, the next example is essentially redundant:

```
SAVEPOINT sales_processing ON ROLLBACK RETAIN  
CURSORS ON ROLLBACK RETAIN LOCKS;
```

MySQL

Not supported.

Oracle

Oracle fully supports the ANSI implementation.

```
SAVEPOINT savepoint_name
```

The following example performs several data modifications, and then rolls back to a savepoint:

```
INSERT INTO sales  
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832');  
SAVEPOINT after_insert;  
UPDATE sales SET terms = 'Net 90'  
WHERE sales_id = '7896';  
SAVEPOINT after_update;  
DELETE sales;  
ROLLBACK TO after_insert;
```

PostgreSQL

Not supported.

SQL Server

SQL Server does not support the *SAVEPOINT* command. Instead, it uses the *SAVE* command:

SAVE TRAN[*SACTION*] *savepoint_name*

In addition, rather than declaring the literal name of the savepoint, you can optionally reference a variable containing the name of the savepoint. If you use a variable, it must be of the *CHAR*, *VARCHAR*, *NCHAR*, or *NVARCHAR* datatype.

SQL Server allows you to have many different named savepoints in a single transaction. However, be careful. Since SQL Server supports multiple savepoints in a single transaction, it might appear that SQL Server fully supports nested savepoints. In fact, it *does not*. Any time you issue a commit or savepoint in SQL Server, it only commits or rolls back to the last open savepoint.

When the *ROLLBACK TRAN savepoint_name* command is executed, SQL Server rolls the transaction back to the specified savepoint, then continues processing with the next valid Transact-SQL command following the *ROLLBACK* statement. Finally, the transaction must ultimately be concluded with a *COMMIT* or a final *ROLLBACK* statement.

See Also

COMMIT
ROLLBACK

SELECT Statement

The *SELECT* statement retrieves rows, columns, and derived values from one or many tables of a database.

Platform	Command
DB2	Supported, with variations (ANSI joins supported)
MySQL	Supported, with variations (ANSI joins partially supported)
Oracle	Supported, with variations (ANSI joins supported)
PostgreSQL	Supported, with variations (ANSI joins partially supported)
SQL Server	Supported, with variations (ANSI joins supported)

SQL2003 Syntax

The full syntax of the *SELECT* statement is powerful and complex, but can be broken down into these main clauses:

SELECT [{*ALL* | *DISTINCT*}] *select_item* [*AS alias*] [...]

FROM [*ONLY* | *OUTER*]

 {*table_name* [[*AS alias*] | *view_name* [[*AS alias*]]} [...]

[[*join_type*] *JOIN join_condition*]

[*WHERE search_condition*] [{*AND* | *OR* | *NOT*} *search_condition* [...]]

[GROUP BY *group_by_expression*{*group_by_columns* | ROLLUP *group_by_columns* |
CUBE *group_by_columns* | GROUPING SETS (*grouping_set_list*) |
() | *grouping_set* , *grouping_set_list*}
[HAVING *search_condition*]]
[ORDER BY {*order_expression* [ASC | DESC]} [,...]]

Keywords

Each of the keywords shown below, except the *select_item* clause, is discussed in greater detail in the "Rules at a Glance" section that follows:

[ALL | DISTINCT] *select_item*

Retrieves values that compose the query result set. Each *select_item* may be a literal, an aggregate or scalar function, a mathematic calculation, a parameter or variable, or a subquery, but a *select_item* is most commonly a column from a table or view. A comma must separate each item in a list of such items.

The schema or owner name should be prefixed to a column when extracted from a context outside of the current user. If the table is owned by another user, then that user must be included in the column reference. For example, assume the user **jake** needs to access data in the schema **katie**:

```
SELECT emp_id  
FROM katie.employee;
```

You can use the asterisk (*) shorthand to retrieve all columns in every table or view shown in the *FROM* clause. It's a good idea to use this shortcut on single-table queries only.

ALL, the default behavior, returns all records that meet the selection criteria. *DISTINCT* tells the database to filter out any duplicate records, thus retrieving only one instance out of many identical records.

AS *alias*

Replaces a column heading (when in the *select_item* clause) or a table name or view name (when in the *FROM* clause) with a shorter heading or name. This clause is especially useful for replacing cryptic or lengthy names with short, easy to understand names or mnemonics, or when the column is only derived data so you don't end up with a column called **ORA000189x7/0.02**. It is also very useful in self-joins and correlated subqueries where a single query references the same table more than once. When multiple items appear in the *select_item* clause or *FROM* clause, make sure to place the commas *after* an *AS alias* clauses. Be careful to always use an alias uniformly once you introduce it into the query.

FROM *table_name*

Serves to list all tables and/or views from which the query retrieves data. Separate table and view names using commas. The *FROM* clause also allows you to assign aliases to long table/or view names or subqueries using the *AS* clause. Using shorter aliases instead of longer table or view names simplifies coding. (Of course, this might thwart the DBA's carefully planned naming conventions, but the alias only lasts for the duration of the query. Refer to the following section on "Rules at a Glance.") *FROM* clauses may contain a subquery (refer to the [SUBQUERY Substatement](#) section).

ONLY

Specifies that only the rows of the named table or view, and no rows in subtables or subviews, will be retrieved in the result set. When using *ONLY*, be sure to enclose the *table_name* or *view_name* within parentheses. *ONLY* is ignored if the table or view has no subtables or subviews.

OUTER

Specifies that the rows of the named table or view, along with the rows and columns of any and all subtables or subviews, will be retrieved in the result set. Columns of the subtables (or subviews) will be appended to the right, in subtable hierarchy order according to depth of the subtable. In extensive hierarchies, subtables with common parents are appended in creation order of their types. When using *OUTER*, be sure to enclose the *table_name* or *view_name* within parentheses. *OUTER* is ignored if the table or view has no subtables or subviews.

JOIN *join_condition*

Joins together the result set of the table shown in the *FROM* clause to another table that shares a meaningful relationship based on a common set of values. These values are usually contained in columns of the same name and datatype that appear in both tables being joined. These columns, or possibly a single column from each table, are called the *join key* or *common key*. Most—but not all—of the time, the join key is the primary key of one table and a foreign key in another table. As long as the data in the columns match, the join can be performed. (Note that joins can also be performed using the *WHERE* clause. This technique is sometimes called a *theta join*.)

Join_conditions are most commonly depicted in the form:

```
JOIN table_name2 ON table_name1.column1 <comparison operator>
```

```
table_name2.column2
```

```
JOIN table_name3 ON table_name1.columnA <comparison operator>
```

```
table_name3.columnA
```

```
[...]
```

When the *comparison_operator* is the equal sign (=), a join is said to be an *equijoin*. However, the comparison operation could be any acceptable comparison operation like <, >, <=, >=, or even <>.

Use the *AND* operator to issue a *JOIN* with multiple conditions. You can also use the *OR* operator to specify alternate join conditions.

If an explicit *join_type* is omitted, then an *INNER JOIN* is assumed. Note that there are many types of joins, each with their own rules and behaviors, described in the "Rules at a Glance" (and in more detail in the [JOIN Subclause](#) section). Note that an alternate approach to the join condition, via the *USING* clause, exists:

```
USING (column_name [,...])
```

Acts as an alternative to the *ON* clause. Instead of describing the conditions of the join, simply provide a *column_name* (or column names separated by commas) that appears in both tables. The database then evaluates the join based on the column (or columns) of *column_name* that appear in both tables. (The column names must be identical in both tables.) In the following example, the two queries produce identical results:

```
SELECT emp_id
```

```
FROM employee
```

```
LEFT JOIN sales USING (emp_id, region_id);
```

```
SELECT emp_id
```

```
FROM employee AS e
```

```
LEFT JOIN sales AS s
```

```
ON e.emp_id = s.emp_id
```

```
AND e.region_id = s.region_id;
```

WHERE *search_condition*

Filters unwanted data from the result set of the query, returning only those records that satisfy the search conditions. A poorly written *WHERE* clause can ruin the performance of an otherwise useful *SELECT* statement, so the nuances of the *WHERE* clause *must be mastered* thoroughly. *Search_conditions* are syntactically depicted in the form:

WHERE [*schema*.*table_name*.]*column operator value*

WHERE clauses usually compare the values contained in a *column* of the table. The values of the column are compared using an *operator* of some type (refer to [Chapter 2](#) for more detail). For example, a column might equal (=) a given *value*, be greater than (>) a given *value*, or be *BETWEEN* a range of *values*.

WHERE clauses may contain many search conditions concatenated together using the *AND* or *OR* Boolean operators. Parentheses can be used to impact the order of precedence of the search conditions. *WHERE* clauses can contain subqueries (refer to the [SUBQUERY Substatement](#) section).

GROUP BY *group_by_expression*

Used in queries that utilize aggregate functions such as *AVG*, *COUNT*, *COUNT DISTINCT*, *MAX*, *MIN*, and *SUM* to group result sets into the categories you define in the *group_by_expression*. The *group_by_expression* of the *GROUP BY* clause has an elaborate syntax of its own:

[*GROUP BY* *group_by_expression*

where *group_by_expression* is:

```
{ (grouping_column [...]) | ROLLUP (grouping_column [...]) |  
  CUBE (grouping_column [...]) | GROUPING SETS ( grouping_set_list ) |  
  ( ) | grouping_set , grouping_set_list}
```

Refer below to the subsection [The GROUP BY clause](#) in the "Rules at a Glance" section for examples and more information on *ROLLUP*, *CUBE*, and *GROUPING SETS*.

HAVING *search_condition*

Adds search conditions on the results of the *GROUP BY* clause in a manner similar to the *WHERE* clause. *HAVING* does not affect the rows used to calculate the aggregates. *HAVING* clauses may contain a subquery. (Refer to the [SUBQUERY Substatement](#) section for more information).

ORDER BY *order_expression* [ASC | DESC]

Sorts the result set in either ascending (ASC) or descending (DESC) order using the *order_expression* specified. The *order_expression* is a list of columns, separated by commas, that you want to sort by.

Rules at a Glance

Each clause of the *SELECT* statement has a specific use. Thus, it is possible to speak individually of the *FROM* clause, the *WHERE* clause, the *GROUP BY* clause, and so forth. You can get more details and examples of *SELECT* statements by looking up the entries for each clause of the statement. However, not every query needs every clause. At a minimum, a query needs a *SELECT* item list and a *FROM* clause. Because the *SELECT* clause is so important and offers so many options, we've divided "Rules at a Glance" into the following detailed subsections:

- Aliases and *WHERE* clause joins
- The *JOIN* clause
- The *WHERE* clause
- The *GROUP BY* clause
- The *HAVING* clause
- The *ORDER BY* clause

Aliases and *WHERE* clause joins

Columns may need to be prefixed with the database, schema, and table name, particularly when there may be the same column name in one or more tables of the query. For example, (on an Oracle database) both the **jobs** table and **scott's employee** table contain the **job_id** column. Note that the following examples join the **employee** and **jobs** table using the *WHERE* clause:

```
SELECT  scott.employee.emp_id,
        scott.employee.fname,
        scott.employee.lname,
        jobs.job_desc
FROM    scott.employee,
        jobs
WHERE   scott.employee.job_id = jobs.job_id
ORDER BY scott.employee.fname,
        scott.employee.lname
```

You can also use aliases to write a query more simply and easily:

```
SELECT  e.emp_id,
        e.fname,
        e.lname,
        j.job_desc
FROM    scott.employee AS e,
        jobs AS j
WHERE   e.job_id = j.job_id
ORDER BY e.fname,
        e.lname
```

These two queries also illustrate some important rules about *WHERE* clause joins:

1. Use commas to separate multiple elements in a *select_item* list, tables in the *FROM* clause, and items in the *order_expression*.
2. Use the *AS* clause to define aliases.
3. Use aliases consistently throughout the *SELECT* statement once you define them.

In general, you should favor the *JOIN* clause (explained in a moment) over the *WHERE* clause for describing join expressions. This not only keeps your code cleaner, making it easy to differentiate join conditions from search conditions, it also allows you to avoid the possibility of counter-intuitive behavior that results from using the *WHERE* clause for outer joins in some implementations.

The JOIN clause

To perform the same query as in the previous example using an ANSI-style join, list the first table and the keyword *JOIN*, followed by the table to be joined. Once the second table is typed in, type the keyword *ON* and the join condition that would have been used in the old style query. The next example shows the query now in ANSI style.

```
SELECT  e.emp_id, e.fname, e.lname, j.job_desc
FROM    scott.employee AS e
JOIN    jobs AS j ON e.job_id = j.job_id
ORDER BY e.fname, e.lname;
```

Alternately, you could use the *USING* clause. Instead of describing the conditions of the join, simply provide one or more *column_names* (separated by commas) that appears in both of the joined tables. The database then evaluates the join based on the column (or columns) of *column_name* that appear in both tables. (The column names must be identical in both tables.) In the following example, the two queries produce identical results, one using the *ON* clause and one using the *USING* clause:

```
SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);

SELECT emp_id
FROM employee AS e
LEFT JOIN sales AS s
  ON e.emp_id = s.emp_id
  AND e.region_id = s.region_id;
```

You can specify several different types of joins in ANSI style:

Cross join

Specifies the complete cross product of two tables. For each record in the first table, all the records in the second table are joined, creating a potentially *huge* result set. (Of course, the result set would be small if both tables had only four records each, but imagine if they had four million!) This command has the same effect as leaving off the join condition and is also known as a "Cartesian Product." We do not recommend using cross joins.

Inner join

Specifies that unmatched rows in either table of the join should be discarded. If no join type is explicitly defined in the ANSI style, then this is the default.

Left [outer] join

Specifies that all records be returned from the table on the left side of the join statement. If a record returned from the left table has no matching record in the table on the right side of the join, it is still returned. Columns from the right table return NULL values. Many professionals recommend configuring outer joins as left joins wherever possible for consistency.

Right [outer] join

Specifies that all records be returned from the table on the right side of the join statement, even if the table on the left has no matching record. Columns from the left table return NULL values.

Full [outer] join

Specifies that all rows from either table be returned, regardless of matching records in the other table. The result set shows NULL values where no matching data exists in the join.




Not all joins are supported by all platforms, so refer to the [JOIN Subclause](#) section for full details on platform-specific join support.

The WHERE clause

A poorly written *WHERE* clause can ruin an otherwise beautiful *SELECT* statement, so the nuances of the *WHERE* clause *must be mastered* thoroughly. This is an example of a typical query and a multipart *WHERE* clause:

```
SELECT  a.au_lname,  
        a.au_fname,  
        t2.title,  
        t2.pubdate  
FROM    authors a  
JOIN    titleauthor t1 ON a.au_id = t1.au_id  
JOIN    titles t2 ON t1.title_id = t2.title_id  
WHERE   (t2.type = 'business' OR t2.type = 'popular_comp')  
        AND t2.advance > 5500  
ORDER BY t2.title
```

In examining this query, the parentheses impact the order of processing for the search conditions. Use parentheses to move search conditions up or down in precedence just like you would in an algebra equation.

	On some platforms, the database's default collation (also known as <i>sort order</i>) impacts how the <i>WHERE</i> clause filters results for a query. For example, SQL Server is (by default) <i>dictionary-order</i> and <i>case-insensitive</i> , making no differentiation between "Smith," "smith," and "SMITH." But Oracle uses <i>dictionary-order</i> and is <i>case-sensitive</i> , finding the values "Smith," "smith," and "SMITH" to be unequal.
--	---

The *WHERE* clause offers many more specific capabilities than the preceding example illustrates. The list below references some of the more common capabilities of the *WHERE* clause. Refer to the [WHERE Clause](#) section for full details and examples.

NOT

Inverts a comparison operation using the syntax *WHERE NOT expression*. Thus, you might use *WHERE NOT LIKE...* or *WHERE NOT IN...* in a query.

Comparison operators

Compare any set of values using the operations $<$, $>$, $<>$, $>=$, $<=$, and $=$. For example:

```
WHERE emp_id = '54123'
```

IS NULL or IS NOT NULL conditions

Search for any NULL or NOT NULL values using the syntax *WHERE expression IS [NOT] NULL*.

AND

Merges multiple conditions, returning only those records that meet all conditions, using the *AND* operator. The maximum number of multiple conditions is determined by each platform. For example:

```
WHERE job_id = '12' AND job_status = 'active'
```

OR

Merges alternative conditions, returning records that meet any of the conditions, using the *OR* operator. For example:

```
WHERE job_id = '13' OR job_status = 'active'
```

LIKE

Tells the query to use a pattern-matching string contained within quotation marks. The wildcard symbols supported by each platform are detailed in the [LIKE Operator](#) section. All platforms support the percent sign (%) for a wildcard symbol. For example, to find any phone number starting with the 415 area code:

```
WHERE phone LIKE '415%'
```

EXISTS

Used only with subqueries, *EXISTS* tests to see whether the subquery data exists. It is typically much faster than *WHERE IN* subqueries. For example, the following query finds all authors who are also employees:

```
SELECT au_lname FROM authors WHERE EXISTS (SELECT last_name FROM employees)
```

BETWEEN

Performs a range check to see if a value is in between two values (inclusive of those two values). For example:

```
WHERE ytd_sales BETWEEN 4000 AND 9000.
```

IN

Performs a test to see if an expression matches any one value out of a list of values. The list may be literal, as in *WHERE state IN ('or', 'il', 'tn', 'ak')*. The list of values may also be derived using a subquery:

```
WHERE state IN (SELECT state_abbr FROM territories).
```

SOME | ANY

Functions the same as the *EXISTS* operation, though with a slightly different syntax. For example, the following query finds all authors who are also employees:

```
SELECT au_lname FROM authors WHERE au_lname = SOME(SELECT last_name FROM employees)
```

ALL

Performs a check to see if all records in the subquery match the evaluation criteria. It returns TRUE when the subquery returns zero rows. For example:

```
WHERE city = ALL (SELECT city FROM employees WHERE emp_id = 54123)
```

The GROUP BY clause

The *GROUP BY* clause (and the *HAVING* clause) is needed only in queries that utilize *aggregate functions*.

The *GROUP BY* clause is used to report an aggregated value for one or more rows returned by a *SELECT* statement based on one or more non-aggregated columns called *grouping columns*. For example, here is a query that counts up how many people we hired per year from the year 1999 to 2004:

```
SELECT hire_year, COUNT(emp_id) AS 'nbr_emps'  
  
FROM employee  
  
WHERE status = 'ACTIVE'  
  
    AND hire_year BETWEEN 1999 AND 2004  
  
GROUP BY hire_year;
```

The results are:

```
hire_year  nbr_emps  
-----  
1999      27  
2000      17  
2001      13  
2002      19  
2003      20  
2004      32
```

Queries using aggregate functions provide many types of summary information. The most common aggregate functions include:

AVG

Returns the average of all non-NULL values in the specified column(s).

AVG DISTINCT

Returns the average of all unique, non-NULL values in the specified column(s).

COUNT

Counts the occurrences of all non-NULL values in the specified column(s).

COUNT DISTINCT

Counts the occurrences of all unique, non-NULL values in the specified column(s).

COUNT()*

Counts every record in the table.

MAX

Returns the highest non-NULL value in the specified column(s).

MIN

Returns the lowest non-NULL value in the specified column(s).

SUM

Totals all non-NULL values in the specified column(s).

SUM DISTINCT

Totals all unique, non-NULL values in the specified column(s).

Some queries that use aggregates return a sole value. Single value aggregates are known as a *scalar aggregates*. Scalar aggregates do not need a *GROUP BY* clause. For example:

--Query

```
SELECT AVG(price)
```

```
FROM titles
```

--Results

```
14.77
```

Queries that return both regular column values and aggregate function values are commonly called *vector aggregates*. Vector aggregates use the *GROUP BY* clause and return one or many rows. There are a few rules to follow when using *GROUP BY*:

- Place *GROUP BY* in the proper clause order—after the *WHERE* clause and before the *ORDER BY* clause.
- Include all non-aggregate columns in the *GROUP BY* clause.
- Do not use a column alias in the *GROUP BY* clause, though table aliases are acceptable.

For example, let's suppose you need to know the total purchase amount of several purchases, with an **Order_Details** table that looks like this:

OrderID	ProductID	UnitPrice	Quantity
10248	11	14.0000	12
10248	42	9.8000	10
10248	72	34.8000	5
10249	14	18.6000	9
10249	51	42.4000	40
10250	41	7.7000	10
10250	51	42.4000	35
10250	65	16.8000	15
...			

The following example will give you the results:

```
SELECT OrderID, SUM(UnitPrice * Quantity) AS 'Order Amt'  
FROM order_details  
WHERE orderid IN (10248, 10249, 10250)  
GROUP BY orderid
```

The results are:

OrderID	Order Amt
10248	440.0000
10249	1863.4000
10250	1813.0000

We could further refine the aggregations by using more than one grouping columns. Considering the following query retrieving the average price of our products, grouped first by name and then by size:

```
SELECT name, size, AVG(unit_price) AS 'avg'  
FROM product  
GROUP BY name, size
```

The results are:

Name	Size	avg
Flux Capacitor	small	900
P32 Space Modulator	small	1400
Transmogrifier	medium	1400
Acme Rocket	large	600
Land Speeder	large	6500

In addition, the *GROUP BY* clause supports a number of very important subclauses:

```
GROUP BY [ {ROLLUP | CUBE} ] ( [ grouping_column [,...] ] ) [ , grouping set list]
```

Groups the aggregate values of the result set by one or more grouping columns. (Without *ROLLUP* or *CUBE*, the *GROUP BY* (*grouping_column* [,...]) clause is the most simplest and common form of the *GROUP BY* clause.)

ROLLUP

Produces subtotals for each set of grouping columns as a hierarchical result set. It adds subtotal and grand total rows into the result sets in a hierarchical fashion. *ROLLUP* operations return one row per grouping column where NULL appears in the grouping column to show the subtotaled or totaled aggregate value (illustrated in a moment).

CUBE

Produces subtotals and cross-tabulated totals for all grouping columns. In a sense, the *CUBE* clause enables you to quickly return multidimensional result sets from standard relational tables without a lot of programmatic work. *CUBE* is especially useful when working with large amounts of data. Like *ROLLUP*, *CUBE* provides subtotals of the grouping columns but it also includes subtotal rows for all possible combinations of the grouping columns specified in the query.

```
GROUP BY GROUPING SETS [ {ROLLUP | CUBE} ] ( [ grouping_column [,...] ] ) [ , grouping set list]
```

Enables aggregated groups on several different sets of grouping columns within the same query. This is especially useful when you want to return only a portion of an aggregated result set. The *GROUPING SETS* clause also lets you select which grouping columns to compare, whereas *CUBE* returns all of the grouping columns and *ROLLUP* returns a hierarchical subset of the grouping columns. As the syntax shows, the ANSI standard also allows *GROUPING SETS* to be paired with *ROLLUP* or *CUBE*.

Consider the following clauses and the result sets they return, shown in [Table 3-3](#).

Table 3-3. GROUP BY syntax variations

<i>GROUP BY</i> syntax	Returns the following sets
<i>GROUP BY</i> (col_A, col_B, col_C)	(col_A, col_B, col_C)
<i>GROUP BY ROLLUP</i> (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) ()
<i>GROUP BY CUBE</i> (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) (col_B, col_C) (col_B) (col_C) ()
<i>GROUP BY GROUPING SETS</i> ((col_A, col_B), (col_A, col_C), (col_C))	Subquery: SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')

Each type of *GROUP BY* clause returns a different set of aggregated values and, in the case of *ROLLUP* and *CUBE*, totals and subtotals.

The concepts of *ROLLUP*, *CUBE*, and *GROUPING SETS* are much more intuitive when explained by example. In the following example, we query for data summarizing the number of **sales_orders** by **order_year** and by **order_quarter**:

```
SELECT order_year AS year, order_quarter AS quarter,
       COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY ROLLUP (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```


The results are:

year quarter orders

```
NULL NULL 648 -- the grand total
2003 NULL 380 -- the total for year 2003
2003 1 87
2003 2 77
2003 3 91
2003 4 125
2004 NULL 268 -- the total for year 2004
2004 1 139
2004 2 119
2004 3 10
```

Adding grouping columns to the query provides more details (and more subtotaling) in the result set. We'll modify the previous example by adding a **region** to the query (but since the number of rows increases, we'll only look at the first and second quarter):

```
SELECT order_year AS year, order_quarter AS quarter, region,
       COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
      AND order_quarter IN (1,2)
      AND region IN ('USA', 'CANADA')
GROUP BY ROLLUP (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

The results are:

year quarter region orders

```
NULL NULL NULL 183 -- the grand total
2003 NULL NULL 68 -- the subtotal for year 2003
2003 1 NULL 36 -- the subtotal for all regions in q1 of 2003
2003 1 CANADA 3
2003 1 USA 33
2003 2 NULL 32 -- the subtotal for all regions in q2 of 2004
2003 2 CANADA 3
2003 2 USA 29
2004 NULL NULL 115 -- the subtotal for year 2004
2004 1 NULL 57 -- the subtotal for all regions in q1 of 2004
```

```
2004 1    CANADA  11
2004 1    USA      46
2004 2    NULL    58    -- the subtotal for all regions in q2 of 2004
2004 2    CANADA   4
2004 2    USA     54
```

The *GROUP BY CUBE* clause is useful for performing multidimensional analysis on aggregated data. Like *GROUP BY ROLLUP*, it returns subtotals combining all of the grouping columns named in the query. (As you will see, it also has the potential to increase the number of rows returned in the result set.)

In the following example, we query for data summarizing the number of **sales_orders** by **order_year** and by **order_quarter**:

```
SELECT order_year AS year, order_quarter AS quarter,
       COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY CUBE (order_year, order_quarter)
ORDER BY order_year, order_quarter;
```

The results are:

```
year quarter orders
```

```
----
```

```
NULL NULL    648    -- the grand total
NULL 1      226    -- the subtotal for q1 of both years
NULL 2      196    -- the subtotal for q2 of both years
NULL 3      101    -- the subtotal for q3 of both years
NULL 4      125    -- the subtotal for q4 of both years
2003 NULL    380    -- the total for year 2003
2003 1       87
2003 2       77
2003 3       91
2003 4      125
2004 NULL    268    -- the total for year 2004
2004 1      139
2004 2      119
2004 3       10
```

The *GROUP BY GROUPING SETS* clause lets you aggregate on more than one group in a single query. For each group set, the query returns subtotals with the grouping column marked as NULL. While the *CUBE* and *ROLLUP* clauses place predefined subtotals into the result set, the *GROUPING SETS* clause allows you to control what subtotals to add to the query. The *GROUPING SETS* clause does not return a grand total.

Using a similar example query to the ones shown with *ROLLUP* and *CUBE*, this time we chose to subtotal by **year** and **quarter** and separately by **year**:

```
SELECT order_year AS year, order_quarter AS quarter, COUNT (*) AS orders
FROM order_details
WHERE order_year IN (2003, 2004)
GROUP BY GROUPING SETS ( (order_year, order_quarter), (order_year) )
ORDER BY order_year, order_quarter;
```

The results are:

year quarter orders

2003 NULL 380 -- the total for year 2003

2003 1 87

2003 2 77

2003 3 91

2003 4 125

2004 NULL 268 -- the total for year 2004

2004 1 139

2004 2 119

2004 3 10

Another way to think of *GROUPING SETS* is to consider them to be like a *UNION ALL* of more than one *GROUP BY* query that references different parts of the same data. You can tell the database to add subtotals to a *GROUPING SET* by simply adding in the *ROLLUP* or *CUBE* clause according to how you would like subtotaling to occur.

GROUPING SETS can also be concatenated to concisely generate large combinations of groupings. *Concatenated GROUPING SETS* yield the cross-product of groupings from each of the sets within a *GROUPING SET* list. Concatenated *GROUPING SETS* are compatible with *CUBE* and *ROLLUP*. Concatenated *GROUPING SETS*, since they perform a cross-product of all *GROUPING SETS*, will generate a very large number of final groupings from even a small number of concatenated groupings.

Table 3-4. GROUP BY syntax variations

GROUP BY syntax	Returns the following sets
GROUP BY (col_A, col_B, col_C)	(col_A, col_B, col_C)
...	...
GROUP BY GROUPING SETS (col_A, col_B) (col_Y, col_Z)	(col_A, col_Y) (col_A, col_Z) (col_B, col_Y) (col_B, col_Z)

In the example in [Table 3-4](#), the concatenated *GROUPING SETS* yield a large number of final groupings. You can imagine how large the result set would be if the concatenated *GROUPING SETS* contained a large number of groupings! However, the information returned can be very valuable and hard to reproduce.

The HAVING clause

The *HAVING* clause adds search conditions on the result of the *GROUP BY* clause. *HAVING* works very much like the *WHERE* clause, but applies to the *GROUP BY* clause. The *HAVING* clause supports all the same search conditions as the *WHERE* clause shown earlier. For example, using the same query as before, we now want to find only those jobs that have more than three people:

--Query

```
SELECT  j.job_desc "Job Description",
        COUNT(e.job_id) "Nbr in Job"
FROM    employee e
JOIN    jobs j ON e.job_id = j.job_id
GROUP BY j.job_desc
HAVING  COUNT(e.job_id) > 3
```

--Results

Job Description	Nbr in Job
Acquisitions Manager	4
Managing Editor	4
Marketing Manager	4
Operations Manager	4
Productions Manager	4
Public Relations Manager	4
Publisher	7

Note that the ANSI standard does not *require* that an explicit *GROUP BY* clause appear with a *HAVING* clause. For example, the following query against the **employee** table is valid because it has an implied *GROUP BY* clause:

```
SELECT COUNT(dept_nbr)
FROM employee
HAVING COUNT(dept_nbr) > 30;
```

Although the earlier code example is valid, this application for the *HAVING* clause is rather rare.

The ORDER BY clause

A result set can be sorted through the *ORDER BY* clause, in accordance with the database's sort order. Each column of the result set may be sorted in either ascending (*ASC*) or descending (*DESC*) order. (Ascending order is the default.) If no *ORDER BY* clause is specified, most implementations return the data according to the physical order of data within the table or according to the order of an index utilized by the query. However, when no *ORDER BY* clause is specified, there is no guarantee on the order of the result set. Following is an example of a *SELECT* statement with an *ORDER BY* clause on SQL Server:

```
SELECT  e.emp_id "Emp ID",
        e.fname "First",
        e.lname "Last",
        j.job_desc "Job Desc"
FROM    employee e,
        jobs j
```

```
WHERE e.job_id = j.job_id
      AND j.job_desc = 'Acquisitions Manager'
ORDER BY e.ename DESC,
         e.lname ASC
```

The results are:

Emp ID	First	Last	Job Desc
MIR38834F	Margaret	Rancé	Acquisitions Manager
MAS70474F	Margaret	Smith	Acquisitions Manager
KJJ92907F	Karla	Jablonski	Acquisitions Manager
GHT50241M	Gary	Thomas	Acquisitions Manager

After the result set is pared down to meet the search conditions, the result set is sorted by the authors' first names in descending order. Where the authors' first names are equal, the results are sorted in ascending order by last name.



You may write an *ORDER BY* clause using columns in the table that do not appear in the select list. For example, you might query all **emp_ids** from the employee table, yet *ORDER BY* the employee's first and last name.

Programming Tips and Gotchas

Once an alias has been assigned to a table or view in the *FROM* clause, use it exclusively for all other references to that table or view within the query (in the *WHERE* clause, for example). Do *not* mix references to the full table name and the alias within a single query. You should avoid mixed references for a couple of reasons. First, it is simply inconsistent and makes code maintenance more difficult. Second, some database platforms return errors on *SELECT* statements containing mixed references. (Refer to the [SUBQUERY Substatement](#) section for special instructions on aliasing within a subquery.)

MySQL, PostgreSQL, and SQL Server support certain types of queries that do not need a *FROM* clause. Use these types of queries with caution since the ANSI standard requires a *FROM* clause. Queries without a *FROM* clause must be manually migrated to either the ANSI standard form or to a form that also works on the target database. Certain platforms do not support the ANSI style *JOIN* clause. Refer to the entry for each clause to fully investigate the varying degrees of support among the different database vendors for all the options of the *SELECT* command.

DB2

DB2 supports all of the typical clauses of the ANSI standard *SELECT* statement, such as *WHERE*, *FROM*, *GROUP BY*, *HAVING*, and *ORDER BY*. DB2 supports all of the subclauses of the *GROUP BY* clause, including concatenated *GROUPING SETS*. Plus, DB2 supplements the most common clauses with its own *SELECT INTO* clause and *FETCH FIRST* clause:

```
SELECT { [ALL | DISTINCT] } select_item [AS alias] [,... ]
[INTO host_variable [,... ] ]
FROM [ONLY | OUTER] {table_name | view_name | TABLE ( {function_name |
    subquery} ) [AS alias] [,... ] }
[ [join type] JOIN table2 ]
[WHERE search_condition] [ {AND | OR | NOT} search_condition [,... ] ]
[GROUP BY group_by_expression [HAVING search_condition] ]
```

[ORDER BY *order_expression* [ASC | DESC]]

[FETCH FIRST {*n*} {ROW | ROWS} ONLY]

where:

INTO *host_variable* [...]

Assigns the values returned by the query to the host variable(s). An error will occur if the *SELECT* statement returns more than one row in the result set. The values of each *select_item* are assigned in ordinal position to the declared *host_variables*. A warning is given if there are fewer host variables than values returned by the *SELECT* statement.

TABLE ({*function_name* | *subquery*})

Specifies that the query will return a function result set or the results of a fully declared subquery. When making a table function call, the column names of the table function specified in the *RETURNS* clause of the *CREATE FUNCTION* statement are analogous the columns of the table. Note that the function call, along with its parameters and the subquery, must be enclosed in parentheses. For example, the following query takes data from an international retail web site and returns the conversions between foreign currencies and the U.S. dollar:

SELECT column1

FROM TABLE (curr_conv(CURRENCY, CONVERSION_RATE)) AS data;

GROUP BY *group_by_expression*

DB2 fully supports the ANSI standard *GROUP BY* clause, including *ROLLUP*, *CUBE*, and *GROUPING SETS*.

FETCH FIRST {*n*} {ROW | ROWS} ONLY

Sets a maximum number of rows that will be retrieved by the query. This clause is used to tell the database that you do not want more than *n* rows, where *n* is any positive integer of 1 or greater, regardless of how many might be retrieved by the database. This can improve performance, since potentially large result sets may be trimmed to a more manageable size.

DB2 supports four possible join types: *INNER*, *LEFT OUTER*, *RIGHT OUTER*, and *FULL OUTER*:

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* with join conditions in the *WHERE* clause is semantically equivalent to an inner join.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all the records in a table even when there are no counterparts in the joined table. For example:

SELECT j.job_id, e.lname

FROM jobs j

LEFT OUTER JOIN employee e ON j.job_id = e.job_id

ORDER BY j.job_id

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. A right join is the same as a left join, except that the optional table is on the left.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

DB2 places some restrictions on using an *ORDER BY* clause or a *FETCH FIRST* clause in any *SELECT* statement used in a subquery. First, a *SELECT* statement containing an *ORDER BY* clause or a *FETCH FIRST* clause must always be enclosed in parentheses, even when used in a set operation. Second, a *SELECT* statement containing an *ORDER BY* clause or a *FETCH FIRST* clause will return an error if it is the primary query of a view or a materialized query table, since views and materialized query tables cannot be inherently ordered.

The *FETCH FIRST* clause can be used to do some interesting things beyond merely limiting the total number of records returned by a result set. For example, the following query does not limit the total number of records returned by the query; instead, it limits only the number of records used in the subquery. In this example, we will return all the titles of the top 10 employees, based on their salaries:

```
SELECT jobs.job_desc
FROM jobs
WHERE jobs.job_desc IN
    (SELECT employee.job_desc
     FROM employee
     ORDER BY salary DESC
     FETCH FIRST 10 ROWS ONLY);
```

In the example above, only the first 10 rows of the employee table will be used, but the parent query might possibly return any number of rows. Also remember that the ordering in the subquery has no impact on the ordering of the final result set.

MySQL

MySQL's implementation of *SELECT* includes partial *JOIN* support, the *INTO* clause, the *LIMIT* clause, and the *PROCEDURE* clause. MySQL does *not* support subqueries prior to Version 4.0.

```
SELECT [DISTINCT |
DISTINCTROW | ALL]
    [STRAIGHT_JOIN] [ {SQL_SMALL_RESULT | SQL_BIG_RESULT} ] [SQL_BUFFER_RESULT]
    [HIGH_PRIORITY]
select_item AS alias [,...]
[INTO {OUTFILE | DUMPFILE} 'file_name' options]
[FROM table_name AS alias [,...]
    [ { USE INDEX (index1 [,...]) | IGNORE INDEX (index1 [,...]) } ]
[join type][JOIN table2] [ON join_condition]
[WHERE search_condition]
[GROUP BY {unsigned_integer | column_name | formula}
    [ASC | DESC] [,...]]
[HAVING search_condition]
[ORDER BY {unsigned_integer | column_name | formula}]
```

[ASC | DESC] [,...]]

[LIMIT [[*offset_position*,] *number_of_rows*]]

[PROCEDURE *procedure_name* (*param* [,...])]

[{FOR UPDATE | LOCK IN SHARE MODE}];

where:

STRAIGHT_JOIN

Forces the optimizer to join tables in the exact order they appear in the *FROM* clause.

SQL_SMALL_RESULT | *SQL_BIG_RESULT*

Tells the optimizer to expect a small or large result set on a *GROUP BY* clause or a *DISTINCT* clause, respectively. MySQL builds a temporary table when a query has a *DISTINCT* or *GROUP BY* clause, and these optional clauses tell MySQL to build a fast temporary table in memory (for *SQL_SMALL_RESULT*) or a slower, disk-based temporary table (for *SQL_BIG_RESULT*) to process the worktable.

SQL_BUFFER_RESULT

Forces the result set into a temporary table so that MySQL can free table locks earlier and speed the result set to the client.

HIGH_PRIORITY

Gives the query a higher priority than statements that modify data within the table. This should be used only for special, high-speed queries.

select_item

Retrieves the expressions or columns listed. Columns may be in the format [*database_name*.] [*table_name*.]*column_name*. If the database or table names are left out, then MySQL assumes the current database and table.

FROM ... USE INDEX | *IGNORE INDEX*

Indicates the table from which rows will be retrieved. The table may be described as [*database_name*.] [*table_name*]. MySQL will treat the query as a join if more than one table appears in the *FROM* clause. The optional *FORCE INDEX* and *USE INDEX* (*index1*, *index2*, ...), and *IGNORE INDEX* (*index1*, *index2*, ...) clauses enable you to tell MySQL to use or ignore specific indexes on a table, respectively.

INTO {OUTFILE | DUMPFILE}'*file_name*'

Writes the result set of the query to a file named '*file_name*' on the host filesystem with the *OUTPUT* option. The *file_name* must not already exist. The *DUMPFIL*E option writes a single continuous line of data without column terminations, line terminations, or escape characters. This option is used mostly for *BLOB* files. Specific rules for using this clause are detailed below.

LIMIT [[*offset_position*,] *number_of_rows*]

Constrains the number of rows returned by the query, starting at the *offset_position* and returning *number_of_rows*. If only one integer is supplied, then the specified number of records is returned, and a default offset (or starting position) of zero is assumed.

PROCEDURE *procedure_name* (*param* [,...])

Names a procedure that processes the data in the result set. The procedure is an external procedure, usually C++, not an internal database stored procedure.

{FOR UPDATE | LOCK IN SHARE MODE}

Issues either a write lock on the rows returned by the query (*FOR UPDATE*) for its exclusive use (provided the table is of InnoDB or BDB type) or issues read locks on the rows returned by the query (*LOCK IN SHARE*) such that other users may see the rows, but may not modify them.

Keep a couple of rules in mind when using the *INTO* clause. First, the output file cannot already exist since overwrite functionality is not supported. Second, any file created by the query will be readable by everyone that can connect to the server. (When using *SELECT...INTO OUTFILE*, you can then turn around and use the MySQL command *LOAD DATA INFILE* to quickly load the data.)

You can use the following options to better control the content of the output file when using *SELECT...INTO OUTFILE*:

- *ESCAPED BY*
- *FIELDS TERMINATED BY*
- *LINES TERMINATED BY*
- *OPTIONALLY ENCLOSED BY*

The following example illustrates the use of these optional commands via a MySQL query that returns a result set in a comma-delimited output file:

```
SELECT job_id, emp_id, lname+fname
INTO OUTFILE "/tmp/employees.text"
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY "\n"
FROM employee;
```

MySQL also allows *SELECT* statements without a *FROM* clause when performing simple arithmetic. For example, the following queries are valid on MySQL:

```
SELECT 2 + 2;
SELECT 565 - 200;
SELECT (365 * 2) * 52;
```

MySQL is very fluid in the way it supports joins. You can use several different syntaxes to perform a join. For example, you can explicitly declare a join in a query using the *JOIN* clause, but then show the join condition in the *WHERE* clause. The other platforms force you to pick one method or the other and do not allow you to mix them in a single query. However, we think it's bad practice to mix methods, so our examples use ANSI *JOIN* syntax.

MySQL supports the following types of *JOIN* syntax:

```
FROM table1, table2
FROM table1 {
  [CROSS] JOIN table2 |
  STRAIGHT_JOIN table2 |
  INNER JOIN table2 [ {ON join_condition |
  USING (column_list)} ] |
  LEFT [OUTER] JOIN table2 {ON join_condition |
  USING (column_list)} |
  NATURAL [LEFT [OUTER]] JOIN table2 |
  RIGHT [OUTER] JOIN table2 [ {ON join_condition |
```

USING (column_list)]]

NATURAL [RIGHT [OUTER]] JOIN table2 }

where:

[CROSS] JOIN table2

Retrieves all records from both *table1* and *table2*. Refer to the [JOIN Subclause](#) section on "Joins" for more information.

STRAIGHT_JOIN table2

Retrieves all records of both *table1* and *table2* as any other *JOIN* statement would, with some special behavior. Normally, MySQL evaluates the table order of a join operation based on choices made by the query optimizer. However, the optimizer can occasionally choose an ineffective table order. The *STRAIGHT_JOIN* keyword forces the optimizer to evaluate the tables in the order listed in the query.

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* is semantically equivalent to an inner join. When using *FROM table1, table2*, be sure include the join condition(s) in the *WHERE* clause.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause that names all columns that exist in both tables. (An example of the *USING* clause appears later.)

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all records in a table even when there are no counterparts in the joined table.

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. A *RIGHT JOIN* is nothing more than a *LEFT JOIN* written from the opposite direction. As a best practice, write all outer join queries as either *LEFT* or *RIGHT OUTER JOIN* queries, but do not use both types, since they require a shift in thinking to understand at first glance.

MySQL offers an interesting alternative to the ANSI SQL standard for querying tables — the *HANDLER* statement. The *HANDLER* statement works a lot like *SELECT*, except that *HANDLER* provides very rapid data reads that circumvent the SQL query engine in MySQL. However, since the *HANDLER* statement is not a SQL statement, refer to the MySQL documentation for more information.

Oracle

Oracle allows a very large number of extensions to the ANSI *SELECT* statement. For example, since both nested tables and partitioned tables are allowed in Oracle (see *CREATE TABLE*), the *SELECT* statement allows queries to those types of structures.

[WITH query_name AS (subquery) [...]]

SELECT { {[ALL | DISTINCT]} | [UNIQUE]} [optimizer_hints]

select_item [AS alias] [...]

[INTO {variable [...] | record}]

FROM {[ONLY] {[schema.][table_name| view_name |

materialized_view_name]}[@database_link]

```
[AS [OF] {SCN | TIMESTAMP} expression] |
subquery [WITH
  {READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]}] |
[[VERSIONS BETWEEN {SCN | TIMESTAMP} {exp | MINVALUE AND
  {exp | MAXVALUE}] AS OF {SCN | TIMESTAMP} expression] |
TABLE (nested_table_column) [(+)]
  {[PARTITION (partition_name) |
  SUBPARTITION (subpartition_name)]}
[SAMPLE [BLOCK] [sample_percentage] [SEED (seed_value)]] [AS alias]
[,...]
[ [join_type] JOIN join_condition [PARTITION BY expression [,...]] ]
[WHERE search_condition]
[ {AND | OR} search_condition [,...]]
[[START WITH value] CONNECT BY [PRIOR] condition] ]
[GROUP BY group_by_expression [HAVING search_condition] ]
[MODEL model_clause]
[ORDER [SIBLINGS] BY order_expression {[ASC | DESC]}
  {[NULLS FIRST | NULLS LAST]}] ]
[FOR UPDATE [OF [schema.][table.]column] [,...]]
  {[NOWAIT | WAIT (integer)]} ]
```

Unless otherwise noted, the clauses shown above follow the ANSI standard. Similarly, elements of the clauses are identical to the ANSI standard unless otherwise noted. For example, Oracle's *GROUP BY* clause is nearly identical to the ANSI standard, including its component elements like *ROLLUP*, *CUBE*, *GROUPING SETS*, concatenated *GROUPING SETS*, and the *HAVING* clause.

WITH *query_name* AS (*subquery*) [,...]

References a named query. Oracle allows you to assign names to subqueries. You can then reference the subquery in multiple places by specifying only its name. (Oracle optimizes this process by treating the subquery as a nested table subquery or a temporary table.)

ALL | DISTINCT | UNIQUE

ALL and DISTINCT are identical to the ANSI *ALL* and *DISTINCT* clauses (see the ANSI SQL syntax earlier). *UNIQUE* is a synonym for *DISTINCT*. *DISTINCT* cannot be used on [LOB](#) columns.

optimizer_hint

Overrides the default behavior of the query optimizer with user-specified behaviors. For example, hints can force Oracle to use an index that it might not otherwise use or to avoid an index that it might otherwise use. Refer to the vendor documentation for more information about optimizer hints.

select_item

May be an expression or a column from a named query, table, view, or materialized view in the format [*schema*.]*[table_name.] column_name*. If you omit the schema, Oracle assumes the context of the current schema. Oracle also allows for *named queries* (explained under the *WITH* keyword above) that may be referenced much like a nested table subquery (refer to the [SUBQUERY Substatement](#) section). Oracle refers to named queries as *subquery factoring*. In addition to named queries, Oracle also supports subqueries and the asterisk (*),

shorthand for all columns, in the *select_item* list.

INTO {*variable* [, ...] | *record*}

Retrieves the result set values into PL/SQL variables or into a PL/SQL record.

FROM [ONLY]

Identifies the table, view, materialized view, partition, or subquery from which the result set is retrieved. The *ONLY* keyword is optional and only applies to views belonging to a hierarchy. Use *ONLY* when you want to retrieve records from a named view only, and not from any of its subviews.

AS [OF] {SCN | TIMESTAMP} *expression*

Implements SQL-driven flashback whereby system change numbers (or timestamps) are applied to each object in the *select_list*. Records retrieved by the query are only those that existed at the specific system change number (SCN) or time. (This feature can also be implemented at a session-level using the *DBMS_FLASHBACK* built-in package.) *SCN expression* must equal a number while *TIMESTAMP expression* must equal a timestamp value. Flashback queries cannot be used on linked servers.

subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT *constraint_name*]}]

Mentioned separately because Oracle allows you extra ways to control a subquery. *WITH READ ONLY* indicates that the target of the subquery cannot be updated. *WITH CHECK OPTION* indicates any update to the target of the subquery must produce rows that would be included in the subquery. *WITH CONSTRAINT* creates a *CHECK OPTION* constraint of *constraint_name* on the table. Note that *WITH CHECK OPTION* and *WITH CONSTRAINT* are usually used in *INSERT...SELECT* statements.

VERSIONS BETWEEN {SCN | TIMESTAMP} {*exp* | MINVALUE AND {*exp* | MAXVALUE}] AS OF {SCN | TIMESTAMP} *expression*]

Specifies a special kind of query using a *flashback_query_clause* to retrieve the history of changes made to data from a table, view, or materialized view. The *VERSIONS_XID* pseudocolumn shows the identifier corresponding to the transaction that made the change. A *flashback_query_clause* requires that you specify an *SCN* (system change number) or a *TIMESTAMP* value for each object in the *select_item* list. (Implement SQL-driven session-level flashback using the Oracle *DBMS_FLASHBACK* package.)

The optional subclause *VERSIONS BETWEEN* is used to retrieve multiple versions of the data specified, using either an upper and lower boundary of an *SCN* (a number) or *TIMESTAMP* (a timestamp value), or by using the *MINVALUE* and *MAXVALUE* keywords. Without this clause, only one past version of the data is returned. (Oracle also provides several version query pseudocolumns for additional versioning information.)

The *AS OF* clause, discussed earlier in this list, determines the SCN or moment in time from which the database issues the query when used with the *VERSIONS* clause.

You cannot use flashback queries with the *VERSIONS* clause against temporary tables, external tables, tables in a cluster, or views.

TABLE

Required when querying a hierarchically declared nested table.

PARTITION

Restricts a query to the specified partition of the table. In essence, rows are retrieved only from the named partition, not the entire table.

SUBPARTITION

Restricts a query to the specified subpartition of the table. I/O is reduced because the rows are retrieved only from the named subpartition instead of the entire table.

SAMPLE [BLOCK] [*sampling_percentage*] [SEED (*seed_value*)]

Tells Oracle to select records from a random sampling of rows, as either a percentage of rows or blocks, within the result set rather than from the entire table. *BLOCK* tells Oracle to use block sampling rather than row sampling. The sampling percentage, telling Oracle the total block or row percentage to be included in the sample, may be anywhere between .000001 and 99. The optional *SEED* clause is used to provide limited repeatability. By specifying a seed value, Oracle will attempt to return the same sample from one execution of the query to the next. The seed value can be between 0 and 4294967295. When *SEED* is omitted, the resulting sample will change from one execution of the query to the next. Sampling may be used only on single-table queries.

JOIN

Merges the result sets of two or more tables in a single query. Described in detail below.

PARTITION BY *expression* [...]

Defines a special kind of query called a *partitioned outer join* that extends the conventional outer join syntax by applying a right or left outer join to a partition of one or more rows specified by the *expression*. This is especially useful for querying sparse data along a particular dimension of data thereby returning rows that otherwise would be omitted from the result set. For an example, see the "Oracle" subsection under [JOIN Subclause](#). The *PARTITION BY* clause can be used on either side of an outer join, resulting in a *UNION* of the outer joins of each of the partitions in the partitioned result set and the table on the other side of the join. (When this clause is omitted, Oracle treats the entire result set as a single partition.) *PARTITION BY* is not allowed with *FULL OUTER JOIN*.

WHERE ... [[START WITH *value*] CONNECT BY [PRIOR] *condition*]

Filters records returned in the result set. Oracle allows the use of hierarchical information within tables, whose filtering can be controlled with the *START WITH* clause. *START WITH* identifies the rows that will serve as the parent rows in the result set. *CONNECT BY* identifies the relationship *condition* between the parent rows and their child rows. The *PRIOR* keyword is used to identify the parent rows instead of the child rows.

Hierarchical queries use the *LEVEL* pseudocolumn to identify the root node (1), the child nodes (2), the grandchild nodes (3), and so forth. Other pseudocolumns available in hierarchical queries are *CONNECT_BY_ISCYCLE* and *CONNECT_BY_ISLEAF*. Hierarchical queries are mutually exclusive of the *ORDER BY* and *GROUP BY* clauses. Do not use those clauses in a query containing *START WITH* or *CONNECT BY*. You can order records from siblings of the same parent table by using the *ORDER SIBLINGS BY* clause.

ORDER [SIBLINGS] BY *order_expression* {NULLS FIRST | NULLS LAST}}

Sorts the result set of the query in order of *order_expression*. The *order_expression* may be the column names, their aliases, or integers indicating their ordinal position. The *ORDER SIBLINGS BY* clause is used when querying hierarchical tables with the *CONNECT BY* clause. *ORDER SIBLINGS BY* tells Oracle to preserve any ordering specified in the *CONNECT BY* clause and applies the ordering only to siblings (i.e., rows of equal level in the hierarchy). *NULLS FIRST* and *NULLS LAST* specify that the records containing *NULLS* should appear either first or last, respectively.

FOR UPDATE [OF [*schema*.]...]

Locks the rows of the result set so that other users cannot lock or update them until you're finished with your transaction. *FOR UPDATE* cannot be used in a subquery, in queries using *DISTINCT* or *GROUP BY*, or in queries with set operators or aggregate functions. Child rows in a hierarchical table are not locked when issued against the parent rows. The *OF* keyword is used to lock only the selected table or view. Otherwise, Oracle locks all the tables or views referenced in the *FROM* clause. When using *OF*, the columns are not significant, though real column names and not aliases must be used. The *NOWAIT* and *WAIT* keywords tell Oracle to either return control immediately if a lock already exists or to wait *integer* seconds before returning control to you, respectively. If neither *NOWAIT* nor *WAIT* is specified, Oracle waits until the rows become available.

Unlike some other database platforms, Oracle does not allow a *SELECT* statement without a *FROM* clause. The following query, for example, is invalid:

```
SELECT 2 + 2;
```

In order to work around this, Oracle has provided a special-purpose table called **dual**. Any time you want to write a query that does not retrieve data from a user-created table, such as to perform a calculation, use *FROM dual*. All of the following queries are valid:

```
SELECT 2 + 2  
  
FROM dual;  
  
SELECT (((52-4) * 5) * 8)  
  
FROM dual;
```

Oracle's implementation of *SELECT* is quite straightforward if you want to retrieve data from a table. For example, Oracle allows the use of *named queries*. A named query is, in a sense, an alias to an entire query that can save you time when writing a complex multi-subquery *SELECT* statement. For example:

```
WITH pub_costs AS  
  
    (SELECT pub_id, SUM(job_lvl) dept_total  
  
     FROM employees e  
  
     GROUP BY pub_id),  
avg_costs AS  
  
    (SELECT SUM(dept_total)/COUNT(*) avg  
  
     FROM employeee)  
SELECT * FROM pub_costs  
WHERE dept_total > (SELECT avg FROM avg_cost)  
  
ORDER BY department_name;
```

In this example, we create two named subqueries — **pub_costs** and **avg_costs** — which are later referenced in the main query. The named queries are effectively the same as subqueries; however, subqueries must be written out in their entirety each time they're used while named queries need not.

Oracle allows you to select rows from a single partition of a partitioned table (using the *PARTITION* clause) or to retrieve only a statistical sampling of the rows, as a percentage of rows or blocks, of a results set using *SAMPLE*. For example:

```
SELECT *  
  
FROM sales PARTITION (sales_2004_q3) sales  
  
WHERE sales.qty > 1000;  
  
SELECT *  
  
FROM sales SAMPLE (12);
```

Flashback queries are a feature of Oracle that enable retrieval of point-in-time result sets. For example, you could find out what everyone's salary was yesterday before a big change was applied to the database:

```
SELECT job_lvl, lname, fname  
  
FROM employee  
  
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY);
```

Another interesting Oracle extension of the standard query format is the *hierarchic query*. Hierarchic queries return the results of queries against hierarchically designed tables in the order you define. For example, the following query returns the names of the employees, their position in the hierarchy (represented by the position in the **org_char** column), the employee ID, the manager ID, and their job ID:

```
SELECT LPAD(' ',2*(LEVEL-1)) || lname AS org_chart,  
  
       emp_id, mgr_id, job_id
```

```
FROM employee

START WITH job_id = 'Chief Executive Officer'

CONNECT BY PRIOR emp_id = mgr_id;

ORG_CHART      EMPLOYEE_ID  MANAGER_ID  JOB_ID
-----
Cramer         101          100 Chief Executive Officer
Devon          108          101 Business Operations Mgr
  Thomas       109          108 Acquisitions Manager
  Koskitalo    110          108 Productions Manager
  Tonini       111          108 Operations Manager
Whalen         200          101 Admin Assistant
Chang          203          101 Chief Financial Officer
  Gietz        206          203 Comptroller
Buchanan       102          101 VP Sales
  Callahan    103          102 Marketing Manager
```

In the previous query, the *CONNECT BY* clause defines the hierarchical relationship of the **emp_id** value as the parent row equal to the **mgr_id** value in the child row, while the *START WITH* clause specifies where in the hierarchy the result set should begin.

Oracle supports the following types of *JOIN* syntax (refer to the [JOIN Subclause](#) section for more details):

```
FROM table1 {
  CROSS JOIN table2 |
  INNER JOIN table2 [ {ON join_condition |
    USING (column_list)} ] |
  LEFT [OUTER] JOIN table2 [ {ON join_condition
    | USING (column_list)} ] |
  NATURAL [LEFT [OUTER]] JOIN table2 |
  RIGHT [OUTER] JOIN table2 [ {ON join_condition
    | USING (column_list)} ] |
  NATURAL [RIGHT [OUTER]] JOIN table2
  FULL [OUTER] JOIN table2 }
```

[CROSS] JOIN

Retrieves all records of both *table1* and *table2*. This is syntactically the same as *FROM table1, table2* with no join conditions in the *WHERE* clause.

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* with join conditions in the *WHERE* clause is semantically equivalent to an inner join.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause containing all columns that are in common between the two joined tables. (Be careful if columns have the same name but not the same datatype or the same sort of values!) *LOB* columns cannot be referenced in a natural join. Referencing a *LOB* or collection column in a *NATURAL JOIN* will return an error.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all the records in a table, even when there are no counterparts in the joined table. For example:

```
SELECT j.job_id, e.lname
FROM jobs j
LEFT OUTER JOIN employee e ON j.job_id = e.job_id
ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. A right join is the same as a left join, except that the optional table is on the left.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

ON join_condition

Declares the condition(s) that join the result set of two tables together. This takes the form of declaring the columns in *table1* and *table2* that must match the join condition. When multiple columns must be compared, use the *AND* clause.

USING (column_list)

Acts as an alternative to the *ON* clause. Instead of describing the conditions of the join, simply provide a column name (or columns separated by commas) that appears in both tables. The column names must be identical in both tables and cannot be prefixed with a table name or alias. *USING* cannot be used on *LOB* columns of any type. The following two queries produce identical results. One is written with a *USING* clause and the other specifies join conditions using ANSI syntax:

```
SELECT column1
FROM foo
LEFT JOIN poo USING (column1, column2);
```

```
SELECT column1
FROM foo
LEFT JOIN poo ON foo.column1 = poo.column1
AND foo.column2 = poo.column2;
```

Note that older Oracle syntax for joins centered around the *WHERE* clause and outer joins were described using a (+)

marker. ANSI join syntax only became available with Oracle 9i Release 1. Therefore, you will often see existing code using the old syntax. For example, the following query is semantically equivalent to the earlier example of a left outer join.

```
SELECT j.job_id, e.lname
FROM jobs j, employee e
WHERE j.job_id = e.job_id (+)
ORDER BY d.job_id
```

This older syntax is sometimes problematic and more difficult to read. You are strongly advised to use the ANSI standard syntax.

Partitioned outer joins

Partitioned outer joins are useful for retrieving sparse data that might otherwise not be easily seen in a result set. (The ANSI standard describes partitioned outer joins, but Oracle is the first to support them.) For example, our **product** table keeps track of all products we produce, while the **manufacturing** table shows when we produce them. Since we're not continuously making every product at all times, the joined data between the two tables may be sparse at times:

```
SELECT manufacturing.time_id AS time, product_name AS name,
       quantity AS qty
FROM product
PARTITION BY (product_name)
RIGHT OUTER JOIN times ON (manufacturing.time_id =
       product.time_id)
WHERE manufacturing.time_id
       BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
       AND TO_DATE('06/10/05', 'DD/MM/YY')
ORDER BY 2, 1;
```

returns the following:

time	name	qty
01-OCT-05	flux capacitor	10
02-OCT-05	flux capacitor	
03-OCT-05	flux capacitor	
04-OCT-05	flux capacitor	
05-OCT-05	flux capacitor	
06-OCT-05	flux capacitor	10
06-OCT-05	flux capacitor	8
01-OCT-05	transmogrifier	10
01-OCT-05	transmogrifier	15
02-OCT-05	transmogrifier	
03-OCT-05	transmogrifier	
04-OCT-05	transmogrifier	10

04-OCT-05 transmogrifier 11

05-OCT-05 transmogrifier

06-OCT-05 transmogrifier

The example query and result set show that partitioned outer joins are useful for retrieving result sets that might otherwise be hard to query due to sparse data.

Flashback queries

Oracle 10g also supports flashback queries — queries that keep track of previous values of the result requested by the *SELECT* statement. In the following set of example code, we'll issue a regular query on a table, change the values in the table with an *UPDATE* statement, and then query the flashback version of the data. First, the regular query:

```
SELECT salary FROM employees  
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY  
-----  
3800
```

Now, we'll change the value in the **employees** table and query the table to confirm the current value:

```
UPDATE employees SET salary = 4000  
WHERE last_name = 'McCreary';  
SELECT salary FROM employees  
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY  
-----  
4000
```

Finally, we'll perform a flashback query to see what the **salary** value was in the past:

```
SELECT salary FROM employees  
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY)  
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY  
-----  
3800
```

If we wanted to be more elaborate, we could find out all of the values of **salary** for a given time period, say, the last

two days:

```
SELECT salary FROM employees

VERSIONS BETWEEN TIMESTAMP

    SYSTIMESTAMP - INTERVAL '1' MINUTE AND

    SYSTIMESTAMP - INTERVAL '2' DAY

WHERE last_name = 'McCreary';
```

The results are:

```
SALARY
-----
4000
3800
```

The MODEL clause

Oracle Database 10g features a powerful new clause, called *MODEL*, which enables spreadsheet-like result sets from a *SELECT* statement. The *MODEL* clause, in particular, is designed to alleviate the need for developers to extract data from the database and put it into a spreadsheet, like Microsoft Excel, for further manipulation. It creates a multidimensional array in which cells can be referenced by dimension values. For instance, you might dimension an array on product and time, specifying column values that you wish to access via combinations of those two dimensions. You can then write rules that are similar in concept to spreadsheet formulas, that are executed in order to change values in your model, or that create new values, and perhaps even new rows, in your model.

Syntactically, the *MODEL* clause appears after the *GROUP BY* and *HAVING* clauses and before the *ORDER BY* clause. The earlier syntax diagram for Oracle's *SELECT* statement shows the position of the clause, and the syntax details are presented here:

MODEL model_clause

model_clause ::= [options] [return] [reference_models] main_model

options ::= [{IGNORE|KEEP} NAV] [UNIQUE {DIMENSION|SINGLE REFERENCE}]

return ::= RETURN {UPDATED|ALL} ROWS

reference_models ::= reference_model [reference_model...]

reference_model ::= REFERENCE model_name ON (subquery) model_columns

[options]

model_columns ::= [partition [alias]]

DIMENSION BY (column[, column...])

MEASURES (column[, column...])

partition ::= PARTITION BY (column[, column...])

column ::= {column_name|expression} [AS alias]

main_model ::= [MAIN model_name] model_columns [options] rules

rules ::= [RULES [UPSERT|UPDATE] [{AUTOMATIC|SEQUENTIAL} ORDER]]

[ITERATE (count) [UNTIL (termination_condition)]]

(rule[, rule...])

rule ::= [UPSERT|UPDATE] cell_reference [ordering] = expression

cell_reference ::= measure[*multi_column_for_loop*][*dimension_indexes*]

multi_column_for_loop ::= FOR (*dimension*[, *dimension*...]) IN (*dimension_values*)

dimension_values ::= (*{subquery}dimension_value*[, *dimension_value*...])

dimension_value ::= (*literal*[, *literal*...])

dimension_indexes ::= *index*[, *index*...]

index ::= *{condition}expression**single_column_for_loop*

single_column_for_loop ::= FOR *dimension* *{in_for_options}like_for_options*

in_for_options ::= IN (*{subquery}literal*[, *literal*...])

like_for_options ::= [LIKE *pattern*] FROM *start_literal* TO *end_literal*

{INCREMENT|DECREMENT} diff_literal

ordering ::= ORDER BY (*order_column*[, *order_column*...])

order_column ::= *{expression}alias* [ASC|DESC] [NULLS FIRST|NULLS_LAST]

The parameters of this *MODEL* clause are as follows:

{IGNORE|KEEP} NAV

Specifies whether null or absent values (*NAV*) are retained as nulls (*KEEP*), or whether they are replaced with suitable defaults (*IGNORE*): zero for numeric types, 1-Jan-2000 for date types, an empty string for character types, and null for anything else.

UNIQUE {DIMENSION|SINGLE REFERENCE}

Specifies the scope within which the database ensures that a given cell reference points to a unique data value. Use *DIMENSION* to require that each *possible* cell reference, whether on the left or right side of a rule, represent a single value. Use *SINGLE REFERENCE* to perform that check only for those cell references that appear on the right-hand side of a rule.

RETURN {UPDATED|ALL} ROWS

Specifies whether all rows are returned from model processing, or whether only updated row are returned.

reference_model

Defines a reference model on which you cannot perform calculations, but containing values that you can reference from within your main query.

model_name ON (*subquery*)

Specifies the name and rowsource for a reference model.

alias

Specifies an alias for a partition.

DIMENSION BY (column[, *column*...])

Specifies the dimensions for a model. Values from these columns represent the set of index values that are used to identify cells in the multidimensional addressing space.

MEASURES (*column*[, *column*...])

Specifies the values associated with each unique combination of dimensions (e.g., with each cell of the model).

PARTITION BY [(]*column*[, *column*...][])

Splits a model into independent partitions based on the columns given. You can not partition reference models.

main_model

Represents the model on which you perform work. Rows from your containing *SELECT* feed into this model, rules are applied, and the resulting rows are returned.

MAIN *model_name*

Begins the definition of the main model, and also gives that model a name.

RULES [*UPSERT*|*UPDATE*]

Specifies whether rules may both create new cells and update existing cells (*UPSERT*), or whether they much only update existing cells (*UPDATE*). If you want your model to be able to create new rows in your result set, specify *UPSERT*. The default is *UPSERT*. You can also control this behavior on a rule-by-rule basis; see *rule* in the syntax.

{*AUTOMATIC*|*SEQUENTIAL*} ORDER

Specifies whether the optimizer determines the order in which rules are evaluated (*AUTOMATIC*), or whether rules are evaluated in the order in which you list them (*SEQUENTIAL*). The default is *SEQUENTIAL*.

ITERATE (*count*)

Requests that entire set of rules be evaluated repeatedly, *count* times. The default is to evaluate the set of rules just once.

UNTIL (*termination_condition*)

Specifies a condition that, when met, causes iteration to cease. You must still specify a *count*, which serves as somewhat of a safeguard against infinite loops.

measure[...]

Is a reference to one of the measures listed in the *MEASURES* clause. When you reference a measure, the square-brackets are part of the syntax. You must specify all dimensions, either via a subquery or by listing them, and the specific value of the measure associated with those dimensions will be returned, or referenced.

multi_column_for_loop

A for-loop iterating over multiple dimensions at one time. The easiest way to picture this is to think of a subquery from which each row represents a specific combination of dimensions.

dimension_indexes

A list of values, whether from columns or expressions, that collectively identify a unique cell in the model.

single_column_for_loop

A for-loop iterating over one dimension.

IN ({*subquery*|*literal*[, *literal*...]})

The source of values for a for-loop may be a subquery, or it may be a specific list of literal values.

LIKE *pattern*

Allows you to insert dimension values into a pattern. Use a percent-sign to mark the location at which you want dimension values to be inserted. For example, use `FOR x LIKE 'A%B' FROM 1 TO 3 INCREMENT 1` to generate values such as 'A1B', 'A2B', 'A3B'.

`FROM start_literal TO end_literal {INCREMENT|DECREMENT} diff_literal`

Defines the starting and ending for-loop values, and also the difference between each subsequent value as the loop iterates from start to end.

`ORDER BY (order_column [, order_column...])`

Imposes an order of evaluation with respect to the cells referenced from the left side of a rule. Use this clause if you want a rule to be applied to cells in order. Otherwise, you have no guarantee as to the order in which the rule is applied to the cells that it affects.

Following is a list of functions that have been designed specifically for use in the *MODEL* clause:

`CV()` or `CV(dimension_column)`

Returns the current value of a dimension column. May be used only on the right-hand side of an expression in a rule. When the `CV()` form is used, the dimension column is determined implicitly based on the function call's position in a list of dimension values.

`PRESENTNNV(measure[dimension, dimension...], not_null, was_null)`

Returns either *not_null* or *was_null*, depending on whether the specified measure was null when model processing began. This function may be used only from the right-hand side of a rule expression.

`PRESENTV(measure[dimension, dimension...], did_exist, didnt_exist)`

Returns either *did_exist* or *didnt_exist*, depending on whether the specified measure existed when model processing began. This function may be used only from the right-hand side of a rule expression. Be aware that whether a measure existed is a completely separate question from whether that measure was null.

ITERATION_NUMBER

Returns zero on the first iteration through the rules, 1 on the second iteration, and so forth. This is useful when you want to base rule calculations on the number of iterations.

The following example demonstrates that the *MODEL* clause gives a normal *SELECT* statement the ability to construct a multidimensional array as a result set and calculate inter-row and inter-array values interdependently. The newly calculated values are returned as part of the *SELECT* statement's result set:

```
ELECT SUBSTR(region,1,20) country, SUBSTR(product,1,15) product,
    year, sales
FROM sales_view
WHERE region IN ('USA','UK')
MODEL RETURN UPDATED ROWS
  PARTITION BY (region)
  DIMENSION BY (product, year)
  MEASURES (sale sales)
  RULES (
    sales['Bounce',2006] = sales['Bounce',2005] + sales['Bounce',2004],
    sales['Y Box', 2006] = sales['Y Box', 2005],
    sales['2_Products',2006] = sales['Bounce',2006]
    + sales['Y Box',2006] )
```

ORDER BY region, product, year;

In this example, a query against the **SALES_VIEW** materialized view returns the sum of sales over the course of a few years for the regions **USA** and **UK**. The *MODEL* clause then falls between the *WHERE* clause and the *ORDER BY* clause. Since **SALES_VIEW** currently holds data for the years 2004 and 2005, we provide it rules to calculate figures for the year 2006.

The subclause *RETURN UPDATED ROWS*, limits the result set to the rows that were created or updated by the query. Next, the example defines the logical divisions of the data using data elements from the materialized view and using the *PARTITION BY*, *DIMENSION BY*, and *MEASURES* subclauses. The *RULES* subclause then references individual measures of the model by referring to combinations of different dimension values much like a spreadsheet macro references worksheet cells with specific lookups and references to ranges of values.

PostgreSQL

PostgreSQL supports a straightforward implementation of the *SELECT* statement. It supports *JOIN* and subquery applications. PostgreSQL also allows the creation of new temporary or permanent tables using the *SELECT...INTO* syntax.

```
SELECT [ALL | DISTINCT [ON (select_item [,...]) ] ]  
  
[AS alias [(alias_list) ] [,...]  
  
[INTO [[TEMP]ORARY] [TABLE] new_table_name]  
  
[FROM [ONLY] table1[.*] [,...]  
  
[ [join type]]JOIN table2 {[ON join_condition] |  
  [USING (column_list)]}  
  
[WHERE search_condition]  
  
[GROUP BY group_by_expression]  
  
[HAVING having_condition]  
  
[ORDER BY order_by_expression [{ASC | DESC  
  | USING operator][,...]  
  
[FOR UPDATE [OF column [,...]  
  
[LIMIT {count | ALL} ][OFFSET [number_of_records]] ]
```

where:

```
[ALL | DISTINCT [ON (select_item [,...]) ] ]
```

Supports the *ALL* and *DISTINCT* keywords of the ANSI SQL standard, where *ALL* (the default) returns all rows (including duplicates) and *DISTINCT* eliminates duplicate rows. In addition, *DISTINCT ON* eliminates duplicates on only one the specified *select_items*, not on all of the *select_items* in the query (example below).

select_item

Includes the standard elements of a *select_item* list supported by the ANSI SQL standard. In addition to the asterisk (*) shorthand to retrieve all rows, you can use *table_name.** to retrieve all rows from an individual table.

```
AS alias [(alias_list) ]
```

Creates an alias or a list of aliases for one or more columns (or tables in the *FROM* clause). *AS* is required for *select_item* aliases, but not for *FROM* table aliases. (Some other database platforms treat the *AS* as an option when declaring an alias.)

```
INTO [[TEMP]ORARY] [TABLE] new_table_name
```

Creates a new table from the result set of the query. Both *TEMP* and *TEMPORARY* are acceptable usages to create a temporary table that is automatically dropped at the end of the session. Otherwise, the command creates a permanent table. Permanent tables created with this statement must have new, unique names but temporary tables may have the same name as an existing table. If you create a temporary table with the same name as an existing permanent table, the temporary table is used to resolve all operations against that table name while in the same session as the one that created it. Other sessions will continue to see the existing permanent table.

FROM [*ONLY*] *table1* [,..]

Specifies one or more source tables where the data resides. (Be sure to specify a join condition or a theta *WHERE* clause so that you don't get a Cartesian product of all records in all tables.) PostgreSQL allows inheritance in child tables of declared parent tables. Use the *ONLY* keyword to suppress rows from the child tables of your source table. (You can turn off this default inheritance globally with the command *SET SQL_Inheritance TO OFF*.) PostgreSQL also supports nested table subqueries (see the [SUBQUERY Substatement](#) section). The *FROM* clause is not needed when used for computation:

```
SELECT 8 * 40;
```

PostgreSQL will also include an implicit *FROM* on *SELECT* statements that include schema-identified columns. For example, the following query is acceptable (though not recommended):

```
SELECT sales.stor_id WHERE sales.stor_id = '6380';
```

GROUP BY *group_by_expression*

Allows *group_by_expressions* that can be the column name or the ordinal number of the column (as determined by its position in the *select_item* list). An example illustrates this concept under the *ORDER BY* entry just below.

ORDER BY *order_by_expression*

Allows *order_by_expressions* that can be the column name, its alias, or the ordinal number of the column (as determined by its position in the *select_item* list.) For example, the following two queries are functionally identical:

```
SELECT stor_id, ord_date, qty AS quantity
```

```
FROM sales
```

```
ORDER BY stor_id, ord_date DESC, qty ASC;
```

```
SELECT stor_id, ord_date, qty
```

```
FROM sales
```

```
ORDER BY 1, 2 DESC, quantity;
```

In single table *SELECT* statements, you may also order by columns of the table that do not appear in the *select_item* list. For example:

```
SELECT *
```

```
FROM sales
```

```
ORDER BY stor_id, qty;
```

ASC and *DESC* are ANSI standard. If not specified, *ASC* is the default. PostgreSQL sorts NULL values as higher than any other value, causing NULL values to appear at the end of *ASC* sorts and at the beginning of *DESC* sorts.

FOR UPDATE OF *column* [,...] LIMIT {*count* | ALL}][OFFSET [*number_of_records*]]]

Limits the number of rows returned by the query to the maximum specified by the integer *count*. The optional *OFFSET* keyword tells PostgreSQL to skip *number_of_records* before starting to return rows. You should make sure you have an *ORDER BY* clause when using *LIMIT* or you may get an unexpected result set. (PostgreSQL 7.0 and greater treat *LIMIT/OFFSET* as a hint and use it to create optimal, possibly very different query plans.)

PostgreSQL supports a handy variation of the *DISTINCT* clause — *DISTINCT ON (select_item [,...])*. This variation allows you to pick and choose the exact columns that are considered for elimination of duplications. PostgreSQL chooses the result set in a manner much like it does for *ORDER BY*. You should include an *ORDER BY* clause so that there's no unpredictability as to which record is returned. For example:

```
SELECT DISTINCT ON (stor_id), ord_date, qty
FROM sales
ORDER BY stor_id, ord_date DESC;
```

The above query retrieves the most recent sales report for each store based on the most recent order date. However, there would be no way to predict what single record would have been returned without the *ORDER BY* clause.

PostgreSQL supports only these types of *JOIN* syntax (refer to the [JOIN Subclause](#) section for more details):

```
FROM table1 [,...] {
    CROSS JOIN table2 |
    [INNER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    LEFT [OUTER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    NATURAL [LEFT [OUTER]] JOIN table2 |
    RIGHT [OUTER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    NATURAL [RIGHT [OUTER]] JOIN table2
    FULL [OUTER] JOIN table2 }
```

where:

[CROSS] JOIN

Retrieves all records of both *table1* and *table2*. This is syntactically the same as *FROM table1, table2*. The results are a Cartesian product (usually a very bad thing!).

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* is semantically equivalent to an inner join without a join condition (i.e., a Cartesian product). When using *FROM table1, table2*, be sure include the join condition(s) in the *WHERE* clause. If you leave out a join type, PostgreSQL assumes you want an inner join.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause that names all columns that have the same name in both tables. (Be careful if columns have the same name but not the same datatype or the same sort of values!) *NATURAL* is mutually exclusive of a specified join condition via *ON* or *USING*.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted. You can use *LEFT OUTER JOIN* to find all

the records in a table even when there are no counterparts in the joined table. For example:

```
SELECT j.job_id, e.lname
FROM jobs j
LEFT OUTER JOIN employee e ON j.job_id = e.job_id
ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there is a matching record in the leftmost table. Analogous to a *LEFT JOIN*.

FULL [OUTER] JOIN

Retrieves all records in the inner join in addition to all records that have no matching records on both the left and right sides.

ON join_condition

Declares the condition(s) that join the result set of two (or more) tables together. This takes the form of declaring the columns in *table1* and *table2* that must be equal. When multiple columns must be compared, use the *AND* clause.

SQL Server

SQL Server supports most of the basic elements of the ANSI *SELECT* statement, including all of the various join types. SQL Server offers several variations on the *SELECT* statement, including optimizer hints, the *INTO* clause, the *TOP* clause, *GROUP BY* variations, *COMPUTE*, and *WITH OPTIONS*.

```
SELECT {[ALL | DISTINCT] | [TOP number [PERCENT]
    [WITH TIES]]} select_item [AS alias]
[INTO new_table_name ]
[FROM {[rowset_function | table1 [,...]] [AS alias]]
[ [join_type]JOIN table2 {[ON join_condition] ]
[WHERE search_condition ]
[GROUP BY {grouping_column [,...]| ALL}] [ WITH { CUBE | ROLLUP } ]
[HAVING search_condition ]
[ORDER BY order_by_expression [ ASC | DESC ] ]
[COMPUTE {aggregation (expression)} [,...]]
    [BY expression [,...]] ]
[FOR {BROWSE | XML { RAW | AUTO | EXPLICIT}
    [, XMLDATA][, ELEMENTS][, BINARY base64] ]
[OPTION ( <hint> [,...]) ]
```

where:

TOP number [PERCENT][WITH TIES]

Specifies that only the *number* of rows are retrieved in the query result set. If *PERCENT* is specified, only the first *number* percent of the rows are retrieved. *WITH TIES* is used only for queries with an *ORDER BY* clause.

This variation specifies that additional rows are returned from the base result set using the same value in the *ORDER BY* clause, appearing as the last of the *TOP* rows.

INTO *new_table_name*

Creates a new table from the result set of the query. You can use this command to create temporary or permanent tables. (Refer to SQL Server's rules for creating temporary or permanent tables in the [CREATE/ALTER TABLE Statement](#) section.) The *SELECT . . . INTO* command quickly copies the rows and columns queried from other table(s) into a new table using a non-logged operation. Since it is not logged, it is not affected by *COMMIT* and *ROLLBACK* statements.

FROM {[*rowset_function* | *table1* [,...]]}

Supports the standard behavior of the ANSI SQL *FROM* clause, including nested table subqueries. In addition, SQL Server supports a set of extensions called *rowset_functions*. Rowset functions allow SQL Server to source data from special or external data sources such as XML streams, full-text search file structures (a special structure in SQL Server used to store things like MS Word documents and MS PowerPoint slide shows within the database), or external data sources (like an MS Excel spreadsheet).

Examples are shown below. (Please see the SQL Server documentation for the full description of the available *FROM* {[*rowset_function* | *table1* [,...]]} options). Among the many possibilities, SQL Server currently supports the following *rowset_functions*:

CONTAINSTABLE

Returns a derived table from a specified table that contains at least one full-text index *TEXT* or *NTEXT* column. The records derived are based upon either a precise, fuzzy, weighted-match, or proximity-match search. The derived table is then treated like any other *FROM* data source.

FREETEXTTABLE

Similar to *CONTAINSTABLE* except that records are derived based upon a meaning search of '*freetext_string*'. *FREETEXTTABLE* is more useful for ad hoc queries against full-text tables, but less accurate than *CONTAINSTABLE*.

OPENDATASOURCE

Provides a means of sourcing data external to SQL Server via OLE DB without declaring a linked server, such as an MS Excel spreadsheet or a Sybase Adaptive Server database table. It is intended for the occasional ad hoc query. If you frequently retrieve result sets from external data sources, then you should declare a linked server.

OPENQUERY

Executes a pass-through query against a linked server. It is an effective means of performing a nested table subquery against a data source that is external to SQL Server. The data source must first be declared as a linked server.

OPENROWSET

Executes a pass-through query against an external data source. It is similar to *OPENDATASOURCE*, except that *OPENDATASOURCE* only opens the data source; it does not actually pass through a *SELECT* statement. *OPENROWSET* is intended for occasional, ad hoc usage only.

OPENXML

Provides a query-able, table-like view to an XML string.

GROUP BY [ALL] (*group_by_expression*) [WITH {CUBE | ROLLUP}]

Supports the ANSI SQL standard with some variations. The first noticeable difference is in syntax. Where the ANSI standard clause is *GROUP BY* [{*CUBE* | *ROLLUP*}] (*grouping_column* [,...]), SQL Server uses *GROUP BY* [ALL] (*group_by_expression*) [WITH {*CUBE* | *ROLLUP*}].

GROUP BY ALL tells SQL Server to provide *group_by* categories even when the matching aggregation is NULL. Normally, SQL Server does not return a category whose aggregation is NULL. It must be used only in conjunction with a *WHERE*

clause. *WITH {CUBE | ROLLUP}* tells SQL Server to perform additional higher-level aggregates of the summary categories. In the simplest terms, *ROLLUP* produces subtotals for the categories, while *CUBE* produces cross-tabulated totals for the categories.



The *GROUPING* function can be used to help differentiate normally occurring NULLs from the NULLs generated by *ROLLUP* and *CUBE* behavior.)

This example groups **royalty** and aggregate **advance** amounts. The *GROUPING* function is applied to the **royalty** column.

USE pubs

```
SELECT royalty, SUM(advance) 'total advance',  
        GROUPING(royalty) 'grp'  
FROM titles  
GROUP BY royalty WITH ROLLUP
```

The result set shows two NULL values under **royalty**. The first NULL represents the group of null values from this column in the table. The second NULL is in the summary row added by the *ROLLUP* operation. The summary row shows the total **advance** amounts for all **royalty** groups and is indicated by 1 in the **grp** column.

Here is the result set:

royalty	total advance	grp
-----	-----	---
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

ORDER BY

Functions normally as specified by the ANSI SQL standard. However, you should remember that SQL Server allows a variety of *collations* that can affect how the result set is evaluated. Thus, under certain collations "SMITH" and "smith" might evaluate and sort differently. Also note that you cannot *ORDER BY* columns of *TEXT*, *NTEXT*, or *IMAGE* datatype.

COMPUTE {*aggregation (expression)*} [,...] [BY *expression* [,...]]]

Generates additional aggregations, usually totals, that appear at the end of the result set. *BY (expression)* adds in subtotals and control breaks to the result set. *COMPUTE* and *COMPUTE BY* can be used simultaneously in the same query. *COMPUTE BY* must be coupled with an *ORDER BY* clause, though the *expression* used by *COMPUTE BY* can be a subset of the *order_by_expression*. The aggregation may be any of these function calls: *AVG*, *COUNT*, *MAX*, *MIN*, *STDEV*, *STDEVP*, *VAR*, *VARP*, and *SUM*. Examples are shown later in this section.

COMPUTE, in any form, does not work with the *DISTINCT* keyword or with *TEXT*, *NTEXT*, or *IMAGE* datatypes.

FOR {*BROWSE* | *XML* { *RAW* | *AUTO* | *EXPLICIT* } [, *XMLDATA*][, *ELEMENTS*][, *BINARY BASE64*]]

FOR BROWSE is used to allow updates to data retrieved in a DB-Library browse mode cursor. (DB-Library is the

original access methodology for SQL Server and has since been supplanted by OLE DB in most applications.)
FOR BROWSE can only be used against tables with a unique index and a column with the *TIMESTAMP* datatype.
FOR BROWSE cannot be used in *UNION* statements or when a *HOLDLOCK* hint is active.

FOR XML

Used to extract the result set as an XML document to the SQL Server 2000 client only. You must further define the resulting XML document as either *RAW*, *AUTO*, or *EXPLICIT*. *RAW* transforms each returned row into a generic XML element with the *<row/>* element tag. *AUTO* transforms the results into a simple, nested XML tree. *EXPLICIT* transforms the resulting XML tree into an explicitly defined shape. However, the query must be written such that the desired nesting information is specified explicitly. You may optionally attach a few extra control features.

XMLDATA

Returns the schema appended to the XML document. *ELEMENTS* returns the columns as subelements instead of mapping them to XML attributes.

BINARY BASE6

Returns the binary data in base64-encoded format. (Binary is the default format for *AUTO* mode, but must be declared for *RAW* and *EXPLICIT*.) *FOR XML* cannot be used in any sort of subquery, in conjunction with a *COMPUTE* clause, *BROWSE* clause, in the definition of a view, in the result set of a user-defined function, or in a cursor. Aggregations and *GROUP BY* are mutually exclusive of *FOR XML AUTO*.

OPTION (*hint* [,...])

Replaces elements of the default query plan with your own. Because the optimizer usually picks the best query plan for any query, you are strongly discouraged from placing optimizer hints into your queries.

Here's an example of SQL Server's *SELECT...INTO* capability. This example creates a table called **non_mgr_employees** using *SELECT...INTO*. The table contains the **emp_id**, first name, and last name of each non-manager from the **employee** table, joined with their job description taken from the **jobs** table:

--QUERY

```
SELECT  e.emp_id, e.fname, e.lname,  
        SUBSTRING(j.job_desc,1,30) AS job_desc  
INTO    non_mgr_employee  
FROM    employee e  
JOIN    jobs AS j ON e.job_id = j.job_id  
WHERE   j.job_desc NOT LIKE '%MANAG%'  
ORDER BY 2,3,1
```

The newly created and loaded table **non_mgr_employee** now can be queried like any other table.



SELECT...INTO should be used only in development or non-production code because it is not logged or recoverable.

Many of SQL Server's extensions to the ANSI *SELECT* statement involve the *GROUP BY* clause. For example, the *GROUP BY ALL* clause causes the aggregation to include NULL valued results to be included when they would normally be excluded. The following two queries are essentially the same except for the *ALL* keyword, yet they produce very different result sets:

```
-- standard GROUP BY
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty <= 10
GROUP BY type
ORDER BY type
-- Results
type      price
-----
business  17.3100
mod_cook  19.9900
popular_comp 20.0000
psychology 13.5040
trad_cook  17.9700
```

Compared to:

```
-- Using GROUP BY ALL
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty = 10
GROUP BY ALL type
ORDER BY type
-- Results
type      price
-----
business  17.3100
mod_cook  19.9900
popular_comp 20.0000
psychology 13.5040
trad_cook  17.9700
UNDECIDED NULL
```

COMPUTE has a number of permutations that can impact the result set retrieved by the query. The following example shows the sum of book prices broken out by type of book and sorted by type and then price:

```
--Query
SELECT type, price
FROM titles
WHERE type IN ('business','psychology')
AND price > 10
```

ORDER BY type, price

COMPUTE SUM(price) BY type

--Results

type price

business 11.9500

business 19.9900

business 19.9900

sum

=====
51.9300

type price

psychology 10.9500

psychology 19.9900

psychology 21.5900

sum

=====
52.5300

The *COMPUTE* clause behaves differently if you do not include *BY*. The following query retrieves the grand total of prices and advances for books over \$16.00:

--Query

SELECT type, price, advance

FROM titles

WHERE price > \$16

COMPUTE SUM(price), SUM(advance)

--Result

type price advance

business 19.9900 5000.0000

business 19.9900 5000.0000

mod_cook 19.9900 .0000

popular_comp 22.9500 7000.0000

popular_comp 20.0000 8000.0000

psychology 21.5900 7000.0000

psychology 19.9900 2000.0000

trad_cook 20.9500 7000.0000

sum

```
=====
165.4500
      sum
=====
41000.0000
```

You could even use *COMPUTE BY* and *COMPUTE* in the same query to produce subtotals and grand totals. (For the sake of brevity, we'll show an example query, but not the result set.) In this example, we find the sum of prices and advances by type for business and psychology books over \$16.00:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
      AND type IN ('business','psychology')
ORDER BY type, price
COMPUTE SUM(price), SUM(advance) BY type
COMPUTE SUM(price), SUM(advance)
```

Don't forget that you must include the *ORDER BY* clause with a *COMPUTE BY* clause! (You do not need an *ORDER BY* clause with a simple *COMPUTE* clause without the *BY* keyword.) There are many permutations you can perform in a single query — multiple *COMPUTE* and *COMPUTE BY* clauses, *GROUP BY* with a *COMPUTE* clause, and even with an *ORDER BY* statement. It's actually fun to tinker around with the different ways you can build queries using *COMPUTE* and *COMPUTE BY*. It's not theme park fun, but what'dya want? This is a programming book!

SQL Server also includes the *FOR XML* clause that converts the standard result set output into an XML document. This is very useful for web database applications. You can execute queries with *FOR XML* directly against the database or within a stored procedure. For example, we can retrieve one of our earlier example queries as an XML document:

```
SELECT type, price, advance
FROM titles
WHERE price > $16
      AND type IN ('business','psychology')
ORDER BY type, price
FOR XML AUTO
```

The results aren't particularly pretty, but they're very usable:

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
```

```
-----
<titles type="business" price="19.9900"
advance="5000.0000"/><titles type="business" price="19.9900" advance="5000.0000"/>
<titles type="psychology" price="19.9900" advance="2000.0000"/><titles
type="psychology" price="21.5900" advance="7000.000
```

If you wanted the XML schema and/or XML elements fully tagged in the output, you could simply append the *XMLDATA* and *ELEMENTS* keywords to the *FOR XML* clause. The query would look like this:


```
SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
FOR XML AUTO, XMLDATA, ELEMENTS
```

SQL Server also implements a number of other enhancements to support XML. For example, the *OPENXML* rowset function can be used to insert an XML document into a SQL Server table. SQL Server also includes system stored procedures that can help you prepare and manipulate XML documents.

SEE ALSO

JOIN
WHERE

SET Statement

The *SET* statement assigns a value to a runtime variable. The variables may be platform-specific system variables or user-defined variables.

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

SET variable = value

Keywords

variable

Denotes a system or user-defined variable.

value

Denotes a string or numeric value appropriate to the system or user-defined variable.

Rules at a Glance

Variable values are set for the duration of the session. The value assigned to the variable must match the datatype of the variable. For example, you cannot assign a string value to a variable that is declared as with a numeric datatype. The actual command to create a variable varies from platform to platform. For example, DB2, Oracle, and SQL Server use the *DECLARE* statement to declare a variable name and datatype, but other platforms may use other means of

creating a variable.

The *value* assigned to a variable does not have to be a literal value. It may be a dynamic value that is derived from a subquery. For example, we assign the maximum employee ID to the **emp_id_var** variable in the following example:

```
DECLARE emp_id_var CHAR(5)

SET emp_id_var = (SELECT MAX(emp_id)

FROM employees WHERE type = 'F')
```

In this example, an employee type of 'F' indicates that the employee is a full-time, salaried employee.

Programming Tips and Gotchas

The *SET* statement is easily transportable between the database platforms. Only Oracle uses a consistently different scheme for assigning a value to a variable. In the following example, we declare a variable on SQL Server called **emp_id_var** and assign a value to it:

```
DECLARE emp_id_var CHAR(5)

SET emp_id_var = '67888'
```

Now, we will perform the same action on an Oracle server:

```
DECLARE emp_id_var CHAR(5);

emp_id_var := '67888';
```

DB2

DB2 supports the basic form of *SET*, assigning values to local variables, output parameters, or special registers. It allows a single *SET* statement to set multiple variable values at one time. It also assigns values to the columns of a base table in a trigger. The command cannot set both types of variables in a single statement.

```
SET variable = { value | NULL | DEFAULT } [...]
```

The DB2 syntax elements are as follows:

variable

Identifies the target variable. SQL variables must be declared before they are used. A variable may also identify a column in the base table of a trigger.

value

Describes the *value* of the variable in a datatype-appropriate format. When assigning values to columns in a trigger, you may also use references to *OLD* and *NEW* correlation names. For more information on this concept, refer to the DB2 platform specifics in the [CREATE/ALTER TRIGGER Statement](#) section.

NULL

Specifies a NULL value for columns or variables that allow NULL values.

DEFAULT

Assigns the default value to columns created with the *WITH DEFAULT* clause or the *IDENTITY* clause. This option assigns NULL to columns that allow NULL and are not defined with a default or identity clause.

To set a single variable value:

```
SET new_var.order_qty = 125;
```

When setting multiple values with a single statement, the number of variables on the left side of the equal sign must equal the number of values on the right side of the equal sign. To set multiple variables in a single statement:

```
SET new_var.order_qty = 125, new_var.discount = 4;
```

When using the *SET variable = SELECT...* variation, the values returned in the result set of the *SELECT* statement must exactly match the variables item for item in ordinal position and datatype. If the *SELECT* statement does not return any rows, then null values are assigned to the variables. You can also use the *SELECT INTO* statement to assign multiple variable values with a single statement.

MySQL

SET, as a keyword in MySQL, has several uses. First, *SET* is a MySQL datatype that allows multiple values each separated by a comma. In addition, *SET* may assign a value to a user variable. That latter use is described here, and the syntax for it is:

```
SET @variable = value [...]
```

When setting multiple values in a single statement, set each value separately with a comma between:

```
SET new_var.order_qty = 125, new_var.discount = 4;
```

In addition, MySQL allows the use of *SELECT* to assign values to variables in the same fashion as that described under the ANSI description of *SET*. However, there are some weaknesses when using the *SELECT* method for assigning values to variables. The primary problem is that values are not assigned immediately within the *SELECT* statement. Thus, for the following example:

```
SELECT (@new_var := row_id) AS a,  
       (@new_var + 3) AS b  
FROM table_name;
```

the **@new_var** variable will not possess the newly selected value of **row_id + 3**. It will retain the value from the beginning of the statement. For this reason, it is a best practice to assign only one value to a variable at a time.

Oracle

The *SET* clause is not supported as the method of assigning values to variables. Instead, user-defined variables are simply assigned a value using the assignment indicator *:=*. The basic syntax is shown here:

```
variable := value
```

PostgreSQL

The PostgreSQL command *SET* is used to set a runtime variable.

```
SET variable { TO | = } { value | DEFAULT }
```

The runtime variable may be set to a string literal designated by *value*. Using the *DEFAULT* keyword sets the runtime variable to its default value. PostgreSQL v7.2 supports these runtime *variables*:

CLIENT_ENCODING NAMES

Sets multibyte client encoding on PostgreSQL installations built with multibyte support

DATESTYLE

Sets the style used to display dates and times. Supported styles include:

ISO

Depicts date and time as YYYY-MM-DD HH:MM:SS (the default ISO 8601 style).

SQL

Depicts the Oracle/Ingres style of date and time, not the date and time style mandated by ANSI SQL.

Postgresql

Depicts the long-time date and time format in PostgreSQL, though no longer the default.

German

Depicts the date and time as dd.mm.yyyy.

You can further qualify the *SQL* and *Postgresql* styles using the keywords as **European**, **US**, and **NonEuropean**, which give dates in the forms dd/mm/yyyy, mm/dd/yyyy, and mm/dd/yyyy, respectively. For example: **SET DATESTYLE = SQL, European**.

SEED

Sets the internal random number generator seed. The value may be any floating-point number between 0 and 1 multiplied by $2^{31}-1$. This value can also be set via the PostgreSQL **setseed** function. For example:

```
SELECT setseed(value);
```

SERVER_ENCODING

Sets multibyte server encoding on servers built with multibyte support.

Here is an example for setting the date and time format to the Oracle and European style:

```
SET DATESTYLE TO sql, European;
```

SQL Server

SQL Server supports *SET* for variable assignments, provided the variables have previously been created with the *DECLARE* statement, and to define values for cursor variables in SQL Server. (SQL Server also uses the *SET* statement for a variety of other purposes, such as enabling or disabling session flags like *SET NOCOUNT ON*.) The platform-specific syntax is:

```
SET { { @variable = value }  
      | { @cursor_variable = { @cursor_variable | cursor_name  
      | { CURSOR [ FORWARD_ONLY | SCROLL ]  
      [ STATIC | KEYSSET | DYNAMIC | FAST_FORWARD ]  
      [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
      [ TYPE_WARNING ]  
FOR select_statement  
      [ FOR { READ ONLY | UPDATE [ OF column_name [,...] ] } }
```

] } } } }

The command does not support the *DEFAULT* keyword, but otherwise supports all the syntax of the ANSI command. The value for *server_name* must reference a connection named in a previous *CONNECT* statement, either as a literal or as a variable.

SET CONNECTION Statement

The *SET CONNECTION* statement allows users to switch between several open connections on one or more database servers.

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

SET CONNECTION {DEFAULT | connection_name}

Keywords

connection_name

Names the connection of the current session. If the *connection_name* is different from the current session connection name, then the connection context is switched to the *connection_name*.

DEFAULT

Switches to the default connection from any other connection. This enables rapid switching to the default connection without knowing its name.

Description

This command does not end a connection. Instead, it switches from the current connection to the connection named (making it the current connection), or from the current connection to the default connection. When switching between connections, the old connection becomes dormant (without committing any changes), while the new connection becomes active.

Rules at a Glance

SET CONNECTION does not create a connection. It merely switches your connection context. Use the *CONNECT* command to create a new connection; use *DISCONNECT* to terminate a connection.

Programming Tips and Gotchas

The *SET CONNECTION* command is not frequently used, since many users connect programmatically via ODBC, JDBC, or some other connectivity method. However, on those platforms that support *SET CONNECTION*, the command can be very useful for rapidly changing connection properties without terminating any existing connections.

DB2

DB2 supports the following basic form of *SET CONNECTION*:

```
SET CONNECTION server_name
```

The DB2 command does not support the *DEFAULT* keyword, but is otherwise compatible with the ANSI command. The value for *server_name* must reference a connection created by a previous *CONNECT* statement.

Following is a fully embedded SQL program in DB2 that shows *CONNECT* and *SET CONNECTION*:

```
EXEC SQL CONNECT TO SQLNUTSHELL;  
  
/* Opens connection to SQLNUTSHELL. The connection will now  
   execute statements referencing objects at SQLNUTSHELL */  
  
EXEC SQL CONNECT TO DB2DEV;  
  
/* Opens connection to SQLNUTSHELL. The connection will now  
   execute statements referencing objects at DB2DEV */  
  
EXEC SQL SET CONNECTION SQLNUTSHELL;  
  
/* Now move back to the SQLNUTSHELL connection and execute  
   statements referencing objects at SQLNUTSHELL. The  
   connection to DB2DEV is still available, only dormant. */
```

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

SQL Server supports *SET CONNECTION*, but only in embedded SQL, not within its ad hoc querying tool, SQL Query Analyzer. Although SQL Server supports the full SQL2003 syntax within SQL embedded in other programs, such as a C++ program, it is not used very often. Most prefer the SQL Server specific command *USE* instead. The platform-specific syntax is:

```
SET CONNECTION connection_name
```

The command does not support the *DEFAULT* keyword, but is otherwise the same as the ANSI command. The value for *connection_name* must reference a connection named in a previous *CONNECT* statement, either as a literal or as a variable.

Here is a full TSQL program in SQL Server that shows *CONNECT*, *DISCONNECT*, and *SET CONNECTION*:

```
EXEC SQL CONNECT TO chicago.pubs AS chicago1 USER sa;  
  
EXEC SQL CONNECT TO new_york.pubs AS new_york1 USER read-only;  
  
-- opens connections to the servers named "chicago" and
```

```
"new_york"  
  
EXEC SQL SET CONNECTION chicago1;  
  
EXEC SQL SELECT name FROM employee INTO :name;  
  
-- sets the chicago1 connection as active and performs work  
    within that session  
  
EXEC SQL SET CONNECTION new_york1;  
  
EXEC SQL SELECT name FROM employee INTO :name;  
  
-- sets the new_york1 connection as active and performs work  
    within that session  
  
EXEC SQL DISCONNECT ALL;  
  
-- Terminates all sessions. You could alternately use two  
-- DISCONNECT commands, one for each named connection.
```

See Also

CONNECT
DISCONNECT

SET CONSTRAINT Statement

The *SET CONSTRAINT* statement defines, for the current transaction, whether a deferrable constraint checks after each DML statement or when the transaction is finally committed. If the session is not currently in an open transaction, the setting applies to the next transaction.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Not supported

SQL2003 Syntax

```
SET CONSTRAINT {constraint_name [,...] | ALL} {DEFERRED | IMMEDIATE}
```

Keywords

{*constraint_name* [,...] | ALL}

Names one or more deferrable constraints where the setting will apply. The keyword *ALL* sets the constraint mode for all deferrable constraints of the current transaction.

DEFERRED

Checks the conditions specified by the deferrable constraint when the transaction is committed, rather than when the DML statements are issued.

IMMEDIATE

Checks the conditions specified by the deferrable constraint immediately after each DML statement is issued, rather than when the transaction is committed.

Rules at a Glance

SET CONSTRAINT defines a value for the constraint mode of all deferrable constraints of the current transaction. If the session is not currently in a transaction, then the *SET CONSTRAINT* statement applies to the next transaction issued during the session.

The following example sets all deferrable constraints to be checked immediately following the issuance of each DML statement:

```
SET CONSTRAINT ALL IMMEDIATE;
```

The next example sets two constraints to defer their data modification until the transaction is committed:

```
SET CONSTRAINT scott.hr_job_title, scott.emp_bonus DEFERRED;
```

Programming Tips and Gotchas

Constraints, when they are defined, may be specified as *DEFERRABLE* or *NOT DEFERRABLE*. The *SET CONSTRAINT* statement will fail if it is issued against a specific constraint that was defined as *NOT DEFERRABLE*.

DB2

Not supported.

MySQL

Not supported.

Oracle

Oracle supports the ANSI standard exactly as described, except that *SET CONSTRAINT* is allowed, as well as *SET CONSTRAINTS*.

PostgreSQL

PostgreSQL supports the ANSI standard syntax exactly as it is described. Currently, PostgreSQL only allows *SET CONSTRAINT* against foreign key constraints, not check and unique constraints, which are always considered *IMMEDIATE*.

SQL Server

Not supported.

SET PATH Statement

The *SET PATH* statement changes the value of the *CURRENT PATH* setting to one or more schemas.

Platform	Command
DB2	Supported, with variations

MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

SET PATH schema_name [,...]

Keywords

schema_name [,...]

Defines one or more schemas as the current path.

Rules at a Glance

SET PATH defines one or more schemas used to qualify an unqualified routine name (that is, functions, procedures, and methods).

The following example sets the current path (i.e., schema name) for unqualified objects to **scott**:

```
SET PATH scott;
```

Then, whenever a routine is referenced during the current session, it will assume the **scott** schema if no schema is identified.

Programming Tips and Gotchas

When referencing multiple schema names, all the schemas must belong to the current database. (The schemas cannot be on a remote database.)

SET PATH does not apply the schema to unqualified objects like tables or views. It only applies to routines.

DB2

DB2 supports several variations on the ANSI standard:

```
SET {[CURRENT] PATH | CURRENT_PATH} =  
    {schema_name | SYSTEM PATH | USER | CURRENT PATH | CURRENT_PATH} [,...]
```

where:

```
SET {[CURRENT] PATH | CURRENT_PATH} =
```

Sets the path, with a slight variation in syntax from the ANSI standard. Note that the equal sign (=) is required.

schema_name

Defines one or more user-specified paths as the current path. The *schema_name* may be a literal, a host variable, or a string constant.

SYSTEM PATH

Defines the system schemas of *SYSIBM*, *SYSFUN*, and *SYSPROC* as the current path.

USER

Defines the value of the *USER* special register as the current path.

{CURRENT PATH | CURRENT_PATH}

Defines the value of the *CURRENT PATH* special register before the *SET PATH* statement was executed as the current path.

The following example sets the schema **scott** and the DB2 system schemas as the current path:

```
SET PATH = scott, SYSTEM PATH
```

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

SET SCHEMA

SET ROLE Statement

The *SET ROLE* statement enables and disables specific roles for the current session.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
SET ROLE {NONE | role_name}
```

Keywords

NONE

Assigns the *CURRENT ROLE* role to the current session.

role_name

Associates the set of privileges associated with the named role with the current session.

Rules at a Glance

When a user session is opened using the *CONNECT* statement, issuing the *SET ROLE* statement grants that session the privileges associated with a role. The *SET ROLE* command can be issued only outside of a transaction.

The value for *role_name* must reference a valid user role existing on the server. You may specify the role name either as a literal or through a variable.

Programming Tips and Gotchas

Sessions are created using the *CONNECT* statement, while roles are created using the *CREATE ROLE* statement.

Most database platforms offer some method of setting or changing the role used during a user session. *SET ROLE* is the ANSI standard approach to setting the role used during a user session, but it is not widely supported by the different database platforms. Check your platform to find an analogous command supported by your specific vendor.

DB2

Not supported. DB2 does not yet support roles.

MySQL

SET ROLE is not supported. An analogous method to control connection settings in MySQL is controlled in the `[client]` section of the `.my.cnf` configuration file in the home directory. For example:

```
[client]
host=server_name
user=user_name
password=client_password
```

To rapidly change between user roles, you can reassign host, user, and password connection properties by assigning new values to `MYSQL_HOST`, `USER` (for Windows only), and `MYSQL_PWD` (though `MYSQL_PWD` is insecure in that other users can view this file), respectively.

Oracle

When a user initiates a connection, Oracle explicitly assigns roles to the user. The role(s) under which the session is operating can be changed with the *SET ROLE* command, assuming the user has permissions to the role. Oracle uses the `MAX_ENABLED_ROLES` initialization parameter (in the `INIT.ORA` file) to control the maximum number of roles that can be enabled concurrently. The Oracle syntax is:

```
SET ROLE { role_name [IDENTIFIED BY password] [,...]
| [ALL [EXCEPT role_name [,...]]
| NONE } ;
```

Options for the *SET ROLE* command include:

role_name

Declares a valid role name already created within Oracle to which the user has been granted permissions. Any roles not specified are unavailable throughout the current session. You can enable multiple roles by providing a list; place a comma between each role listed.

IDENTIFIED BY password

If the *role_name* has a password, that password must be specified using this clause.

ALL

Enables all roles that are granted to the current user, including roles that are granted through other roles, but cannot be used with the *IDENTIFIED BY* clause.

EXCEPT

Specifies a list of roles to exclude from a *SET ROLE ALL* command.

NONE

Disables all roles, including the default role.

Roles with passwords may be accessed only through the statement *SET responsibility role_name IDENTIFIED BY password*. For example, to enable the specific roles **read_only** and **updater**, identified by the passwords **editor** and **red_marker**, respectively, for the current session:

SET ROLE

```
read_only IDENTIFIED BY editor,  
updater IDENTIFIED BY red_marker;
```

To enable all roles except the **read_write** role:

```
SET ROLE ALL EXCEPT read_write;
```

PostgreSQL

SET ROLE is not supported. However, the ANSI SQL command *SET SESSION AUTHORIZATION* is supported and can achieve somewhat similar results.

SQL Server

Not supported.

See Also

CONNECT
DISCONNECT
SET SESSION AUTHORIZATION

SET SCHEMA Statement

The *SET SCHEMA* statement changes the value of the *CURRENT SCHEMA* setting to a user-specified schemas.

Platform	Command
----------	---------

DB2	Supported, with variations
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

SET SCHEMA schema_name [...]

Keywords

schema_name [...]

Defines one or more schema as the current path.

Rules at a Glance

SET SCHEMA defines a user-defined schema to use to qualify an unqualified object, such as a table or view.

The following example sets the current schema for unqualified objects to **scott**:

SET SCHEMA scott;

Then, whenever an object is referenced during the current session, it will assume the **scott** schema, if no schema is identified.

Programming Tips and Gotchas

The *SET SCHEMA* statement cannot assign a schema from a remote database as the *CURRENT SCHEMA*.

SET SCHEMA does not apply the schema to unqualified routines like functions, procedures, and methods. It only applies to database objects like tables and views.

DB2

DB2 supports several variations on the ANSI standard:

SET [CURRENT] SCHEMA = {schema_name | USER } [...]

where:

SET [CURRENT] SCHEMA =

Sets the path, with a slight variation in syntax from the ANSI standard. Note that the equal sign (=) is required.

schema_name

Defines a user-specified schema, of no more than 30 bytes in length, as the current schema. The *schema_name* may be a literal, a host variable, or a string constant.

USER

Defines the value of the USER special register as the current path.

The DB2 implementation is otherwise the same as the ANSI standard.

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

SET PATH

SET SESSION AUTHORIZATION Statement

The *SET SESSION AUTHORIZATION* statement sets the user identifier for the current session.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Not supported

SQL2003 Syntax

SET SESSION AUTHORIZATION *username*

Keywords

username

Sets the session user and the current user of the SQL session to the context of *username*, where *username* may be a literal, a parameter, or a host variable.

Rules at a Glance

This command allows you to switch between users and to run under their permissions.

Programming Tips and Gotchas

Some platforms allow you to use special shortcut keywords, like *SESSION USER* and *CURRENT USER*. *SESSION USER* and *CURRENT USER* are usually the same thing — the username of the currently active session provided by the client. However, *SESSION USER* and *CURRENT USER* can diverge in a session when *SETUID* functions and other similar mechanisms are invoked.

Superuser permissions are required to invoke this command, but you will still be able to switch back to the initial user session even if the current user session does not normally have permission to run *SET SESSION AUTHORIZATION*.

You might also wish to check the value of the *SESSION_USER* and *CURRENT_USER* functions with this SQL statement:

```
SELECT SESSION_USER, CURRENT_USER;
```

Normally, you should issue *SET SESSION AUTHORIZATION* before any transactions to set the session and current user values for all transactions that follow. It must be issued as the only command in its transaction batch.

DB2

Not supported. Similar functionality is achieved with the *CONNECT* statement or by logging off the server and reconnecting.

MySQL

Not supported. You must disconnect your session from MySQL, and then reconnect to utilize another set of user privileges.

Oracle

Not supported. Similar functionality is achieved using the *CONNECT* statement or by logging off the server and reconnecting.

PostgreSQL

PostgreSQL supports the ANSI standard for this command. The only difference, and it is a minor one, is that the ANSI standard does not allow this command during a transaction, while PostgreSQL does not care one way or the other.

SQL Server

Not supported. Similar functionality is achieved with the *CONNECT* statement or by logging off the server and reconnecting.

See Also

- CONNECT
- GRANT
- SESSION_USER

SET TIME ZONE Statement

The *SET TIME ZONE* statement changes the current session's time zone if it needs to be different from the default time zone.

Platform	Command
DB2	Not supported
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Not supported

SQL2003 Syntax

```
SET TIME_ZONE {LOCAL | INTERVAL {+ | -}'00:00' [HOUR TO MINUTE]}
```

Keywords

LOCAL

Sets the current session's time zone to that of the local server.

INTERVAL

Sets an increase (with +) or decrease (with -) over the default time.

INTERVAL

Specifies the time zone offset from Coordinated Universal Time (UTC) in terms of hours and minutes.

HOUR TO MINUTE

Specifies the datatype of the *TIME_ZONE* value.

Rules at a Glance

This is a relatively simple command that sets the user session time zone to that of the server (*LOCAL*), or sets the time zone in relation to Coordinated Universal Time (*UTC*) (formerly Greenwich Mean Time or GMT). Thus, an *INTERVAL* of 2 would advance the time zone two hours greater than UTC, while an *INTERVAL* of -6 would reduce the time zone by six hours from UTC to the United States central time zone.

Programming Tips and Gotchas

Like most *SET* commands, *SET TIME_ZONE* can be executed only outside of an explicit transaction. In other words, you do not need to encapsulate the command within a *START* or *BEGIN TRAN* and a *COMMIT TRAN* statement.

DB2

Not supported.

MySQL

Not supported.

Oracle

In Oracle9i and higher, you can use the following *ALTER SESSION* command to set the session time zone:

```
ALTER SESSION
```

```
  SET TIME_ZONE = {'[+ | -] hh:mm'
```

```
    | LOCAL
```

```
    | DBTIMEZONE
```

```
    | 'region']
```


Use *LOCAL* to revert back to your session's original default time zone. Use *DBTIMEZONE* to set your session time zone to be the database time zone. Use *'region'* to specify a time zone region name such as EST or PST. Use an offset such as *'-5:00'* to specify your time zone in terms of an hour and minute displacement from UTC. A displacement of *'-5:00'* means that your time is five hours behind UTC time (e.g., 5:00 A.M. your time is 10:00 A.M. UTC time).

To retrieve a list of valid time zone region names, issue the following query:

```
SELECT tzone FROM v$timezone_names;
```

Both of the following commands set the time zone to Eastern Standard Time. The first command does this by specifying the appropriate displacement from UTC, while the second command specifies the time zone region name:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
```

```
ALTER SESSION SET TIME_ZONE = 'EST';
```

Oracle's time zone support is complex. Steven Feuerstein's *Oracle PL/SQL Programming* (O'Reilly), contains a good explanation of it in the chapter on datetime datatypes.

PostgreSQL

PostgreSQL allows a session's time value to be set to the server default by interchangeably using the *LOCAL* or *DEFAULT* clauses:

```
SET TIME ZONE {'timezone' | LOCAL | DEFAULT };
```

There are some variations from the ANSI standard:

'timezone'

Specifies a time zone by name. The possible values for time zone depends on the operating system. For example, the file */usr/share/zoneinfo* contains the database of time zones for Linux servers.

LOCAL / DEFAULT

Sets the current-session time values to those of the default time zone as determined by the local server.

For example, *'PST8PDT'* is a valid time zone for California on Linux systems, while *'Europe/Rome'* is a valid time zone for Italy on Linux and other systems. If you specify an invalid time zone, the command sets the time zone to UTC.

If increasing the time zone over UTC, the plus sign is optional. If an invalid time zone is specified, UTC is used as the time zone.

The following example sets the PostgreSQL time zone to Pacific Standard Time:

```
SET TIME ZONE 'PST8PDT';
```

Next, the time for the current session is returned to the server's default time zone:

```
SET TIME ZONE LOCAL;
```

SQL Server

Not supported.

SET TRANSACTION Statement

The *SET TRANSACTION* statement controls many characteristics of data modification, primarily the read/write characteristics and isolation level of a transaction.

Platform	Command
DB2	Not supported
MySQL	Supported, with variations
Oracle	Supported, with limitations
PostgreSQL	Supported
SQL Server	Supported, with variations

SQL2003 Syntax

```
SET [LOCAL] TRANSACTION [READ ONLY | READ WRITE]
    [ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
    REPEATABLE READ | SERIALIZABLE}]
    [DIAGNOSTIC SIZE inf]
```

Keywords

LOCAL

Changes transaction settings for the current session on the local server and only the local server. Otherwise, the transaction settings for the next transaction are changed, even if the transaction runs on a remote server.

READ ONLY

Sets the next upcoming transaction as a read-only transaction. Once the next transaction is complete, transaction behavior reverts to the default settings.

READ WRITE

Sets the next upcoming transaction so it may perform transactions that read and write data.

ISOLATION LEVEL

Sets the isolation level for the next transaction in the session.

READ COMMITTED

Allows a transaction to read rows written by other transactions only when they have been committed.

READ UNCOMMITTED

Allows a transaction to read rows which have been written, but not committed, by other transactions.

REPEATABLE READ

All sessions can see records that are committed before their first transaction was begun. Other open sessions can see or change only committed rows in the user's current session before their transactions were begun. Consequently, later transactions can add records that might then be visible to the transactions of earlier sessions, but the other sessions must query to see those records.

SERIALIZABLE

All sessions can see records that are committed before their transaction was begun. Other open sessions can see but cannot insert or update records within other user sessions until those transaction are completed. This is the most restrictive isolation level and the default for SQL2003.

DIAGNOSTIC SIZE int

Designates the specific number of error messages, *int*, to capture for a transaction. The *GET DIAGNOSTICS* statement retrieves these error messages.

Rules at a Glance

When issued, *SET TRANSACTION* sets the properties of the next upcoming transaction. Because of this, *SET TRANSACTION* is an interim statement that should be issued after one transaction completes and before the next transaction starts. (To begin a transaction and set its characteristics at the same time, use *START TRANSACTION*.) More than one option may be applied with this command but only one access mode, isolation level, and diagnostic size may be specified at a time.

The *isolation level* of a transaction specifies the degree of isolation a transaction has from other concurrently running sessions. The isolation level controls:

- Whether rows read and updated by your database session are available to other concurrently running database sessions.
- Whether the update, read, and write activity of other database sessions can affect your database session.

If you are unfamiliar with isolation levels, be sure to read the [SQL2003 Syntax](#) section under [SET TRANSACTION Statement](#), since the explanation for isolation levels is offered only in that section.

Programming Tips and Gotchas

The *ISOLATION LEVEL* clause controls a number of behaviors and anomalies in a transaction concerning concurrent transactions, particularly the following:

Dirty reads

Occur when a transaction reads the altered records of another transaction before the other transaction has completed. This allows a data modification to occur on a record that might not be committed to the database.

Nonrepeatable reads

Occur when one transaction reads a record while another modifies it. So if the first transaction attempts to reread the record, it can't find it.

Phantom records

Occur when a transaction A reads a group of records, but transaction B adds or changes the data so that more records satisfy the query issued by the transaction A. Thus, transaction A possibly reads in the records of transaction B as if they were committed to the database, when in fact the records from transaction B may still be rolled back. Since transaction A is reading records that are not yet permanent, these are called *phantom records*.

[Table 3-5](#) shows the impact of various isolation level settings on the anomalies just listed.

Table 3-5. Isolation level and anomaly impact

Isolation level	Dirty reads	Nonrepeatable reads	Phantom records
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	Not allowed	Allowed	Allowed

REPEATABLE READ	Not allowed	Not allowed	Allowed
SERIALIZABLE	Not allowed	Not allowed	Not allowed

DB2

Not supported. DB2 supports all isolation levels, but not through the *SET TRANSACTION* statement. Isolation levels are controlled instead through the program preparation process and in server-level settings issued by the database administrator.

Although DB2 supports the ANSI isolation levels, it uses different names for them. DB2 also allows users to set the isolation for specific statements, including *SELECT*, *DELETE*, *INSERT*, and *UPDATE*. DB2 uses the following names for the various isolation levels:

CS

Specifies cursor stability (serializable) as the isolation level.

NC

Specifies no commit as the isolation level. Not supported by DB2.

RR

Specifies repeatable read as the isolation level.

RS

Specifies read stability (read committed) as the isolation level.

UR

Specifies uncommitted read (read uncommitted) as the isolation level.

DB2 also supports two statements, *CHANGE ISOLATION LEVEL* and *SET CURRENT ISOLATION*, that change the way that DB2 isolates data from other processes while a database is being accessed.

MySQL

MySQL allows you to set the transaction isolation level for the next individual transaction, the whole session, or globally across the server, as follows:

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL

[READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ

| SERIALIZABLE]

By default, MySQL sets the isolation level for the transaction that immediately follows the statement.

GLOBAL

Sets the transaction isolation for all subsequent transactions across all user sessions or system threads.

SESSION

Sets the transaction isolation level for all subsequent transactions of the current session.

TRANSACTION ISOLATION LEVEL

Sets a specific transaction isolation level, as described in earlier in the [SET TRANSACTION Statement](#) section. When omitted, MySQL defaults to the *REPEATABLE READ* isolation level.

The *SUPER* privilege is required to set a *GLOBAL* transaction isolation level. You can also set the default isolation level via the *MYSQL* command line executable using the `-transaction-isolation=` switch. Following is an example that sets the all subsequent threads (both user and system threads) to a serializable transaction isolation level:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Oracle

Oracle allows you to set a transaction as read-only or read-write, set the transaction isolation level, or specify a specific rollback segment for your transactions.

```
SET TRANSACTION { [ READ ONLY | READ WRITE ]  
| [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]  
| [ USE ROLLBACK SEGMENT segment_name ]  
| NAME 'transaction_name'};
```

where:

READ ONLY

Sets the next transaction as read-only and serializable. This option is not available to the user **SYS**. The only statements permitted in read-only sessions are *SELECT*, *ALTER SESSION*, *ALTER SYSTEM*, *LOCK TABLE*, and *SET ROLE*.

READ WRITE

The default transaction style in Oracle. It allows transactions to read and write data.

READ COMMITTED

The default transaction isolation level in Oracle. The same as the ANSI standard.

SERIALIZABLE

Sets the transaction isolation level to the ANSI serializable level and requires the *COMPATIBLE* *init* parameter to be set to 7.3.0 or higher.

USE ROLLBACK SEGMENT *segment_name*

Sets the next read-write transaction to be written to a specific Oracle rollback segment identified by *segment_name*. Because it applies only to the current transaction, *USE ROLLBACK SEGMENT* should be the first statement in the transaction. This option is not compatible with the *READ ONLY* option. The rollback segment must already exist or else this statement will fail.

NAME

Assigns a name of 255 characters or less to the current transaction. This option is useful in distributed transaction processing environments, for two-phase commits, because it lets you easily identify which local transactions belong to a single distributed transaction.

The *USE ROLLBACK SEGMENT* variant can be useful for performance tuning by allowing you to direct long-running transactions to rollback segments large enough to hold them, while small transactions could be directed to small rollback segments that might be small enough to be retained in cache.

The *SET TRANSACTION* statement should be the first statement in any SQL batch, though Oracle treats it virtually the

same as the *START TRANSACTION* statement. So one could be substituted for the other.

In the following example, the query reports from a bi-weekly process on the Chicago server while avoiding impact from any other users who might be updating or inserting records:

```
SET TRANSACTION READ ONLY NAME 'chicago';

SELECT prod_id, ord_qty

FROM sales

WHERE stor_id = 5;
```

In another case, late-night batch processing might create a huge transaction that would overflow all but the rollback segment created to support that one transaction:

```
SET TRANSACTION USE ROLLBACK SEGMENT huge_tran_01;
```

PostgreSQL

SET TRANSACTION in PostgreSQL impacts the new transaction you are beginning only. Consequently, you may have to issue this statement before each new transaction.

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | SERIALIZABLE};
```

where:

READ COMMITTED

Sets the transaction isolation level to the ANSI level *READ COMMITTED*. This is the default.

SERIALIZABLE

Sets the transaction isolation level to the ANSI level *SERIALIZABLE*.

By default, PostgreSQL supports the *READ COMMITTED* transaction isolation level. You can set the default transaction isolation level for all transactions in the session by using either of the following commands:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
```

```
{ READ COMMITTED | SERIALIZABLE }
```

```
SET default_transaction_isolation =
```

```
{ 'read committed' | 'serializable' }
```

Of course, you can then override the isolation level of any subsequent transaction using the *SET TRANSACTION* statement.

For example, you can set the next transaction to the serializable transaction isolation level:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Alternately, you could set all the transactions in an entire session to serializable:

```
SET SESSION CHARACTERISTICS AS TRANSACTION
```

```
ISOLATION LEVEL SERIALIZABLE;
```

SQL Server

SET TRANSACTION in SQL Server sets the isolation level for an entire session. All queries that follow a *SET TRANSACTION* statement run under the isolation level set by the statement until it is otherwise changed.

SET TRANSACTION ISOLATION LEVEL

```
{ READ COMMITTED  
| READ UNCOMMITTED  
| REPEATABLE READ  
| SERIALIZABLE}
```

where:

READ COMMITTED

Sets the transaction isolation level to the ANSI level *READ COMMITTED*. This is the default.

READ UNCOMMITTED

Sets the transaction isolation level to the ANSI level *READ UNCOMMITTED*. This has the same effect as the *NOLOCK* optimizer hint.

REPEATABLE READ

Sets the transaction isolation level to the ANSI level *REPEATABLE READ*.

SERIALIZABLE

Sets the transaction isolation level to the ANSI level *SERIALIZABLE*. Similar results can be achieved in SQL Server using the *HOLDLOCK* optimizer hint.

For example, the following command lowers the transaction isolation level for all *SELECT* statements during the session from *READ COMMITTED* to *REPEATABLE READ*:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ  
GO
```

See Also

COMMIT
ROLLBACK

START TRANSACTION Statement

The *START TRANSACTION* statement allows you to perform all the functions of *SET TRANSACTION* while also initiating a new transaction.

Platform	Command
DB2	Not supported
MySQL	Supported, with limitations
Oracle	Not supported
PostgreSQL	Not supported; see <i>BEGIN TRAN</i> below

SQL Server	Not supported; see <i>BEGIN TRAN</i> below
------------	---

SQL2003 Syntax

START TRANSACTION [READ ONLY | READ WRITE]

[ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |

REPEATABLE READ | SERIALIZABLE}]

[DIAGNOSTIC SIZE *int*]

According to the ANSI standard, the only difference between *SET* and *START* is that *SET* is considered outside of the current transaction, while *START* is considered the beginning of a new transaction. Thus, *SET TRANSACTION* settings apply to the next transaction, while *START TRANSACTION* settings apply to the current transaction.

While only MySQL supports the *START TRANSACTION* statement, three of the vendors (MySQL, PostgreSQL, and SQL Server) support a similar command, *BEGIN [TRAN[SACTION]]* and its synonym *BEGIN [WORK]*. *BEGIN TRANSACTION* declares an explicit transaction, but it does not set isolation levels.

Rules at a Glance

The only significant rule of the *START TRANSACTION* statement is that you must use this statement to control the access mode, isolation level, or diagnostic size of the current transaction. Once a new transaction starts, you must either issue new values for the setting(s) or rely on the default.

Most database platforms allow you to implicitly control transactions, using what is commonly called *autocommit* mode. In autocommit mode, the database treats each statement as a transaction in and of itself, complete with an implicit *BEGIN TRAN* and *COMMIT TRAN* statement.

The alternative to autocommit mode is to manually control each transaction explicitly. Under explicit transaction control, you declare each new transaction with the *START TRANSACTION* statement. A new transaction may also start implicitly any time a transaction-initiating statement is issued, such as *INSERT*, *UPDATE*, *DELETE*, or *SELECT*. The transaction is not committed or rolled back until either *COMMIT* or *ROLLBACK* statements are explicitly issued.

Some platforms, namely DB2 and Oracle, do not support the explicit declaration of a new transaction using *START TRANSACTION*, but they do support explicitly committing, savepointing, and rolling back a transaction. Other platforms, such as MySQL, PostgreSQL, and SQL Server, allow you to both explicitly declare a transaction with *START TRANSACTION* and to explicitly commit, savepoint, and roll back the transaction.

Programming Tips and Gotchas

Many of the platforms discussed in this book run in autocommit mode by default. Therefore, it is a good rule of thumb to only use explicitly declared transactions if you intend to do so for all transactions in a session. In other words, do not mix implicitly declared transactions and explicitly declared transactions in a single session. Each transaction that is explicitly declared can only be made permanent with the *COMMIT* statement. Similarly, any transaction that fails or needs to be discarded must be explicitly undone with the *ROLLBACK* statement.



Be sure to issue *START* in a pair with either *COMMIT* or *ROLLBACK*. Otherwise, the DBMS may not complete the transaction(s) until it encounters *COMMIT* or *ROLLBACK*. Omitting timely *COMMITs* (or *ROLLBACKs*) could lead potentially to huge or poorly planned transactions.

It is a good idea to issue an explicit rollback or commit after one or a few statements because long-running transactions can lock up resources, thus preventing other users from accessing those resources. And don't forget to commit (or roll back)! Long-running or very large transaction batches can fill up the rollback segments or transaction logs of a database, even if those files are small.

DB2

Not supported. Transactions in DB2 are started implicitly.

MySQL

MySQL normally runs in autocommit mode, which means changes are automatically saved to disk when completed. If a change fails for any reason, it is automatically rolled back.

MySQL supports *START TRANSACTION*, but it is merely a synonym for *BEGIN*. You can suspend autocommit for one or several statements using the *BEGIN* syntax:

```
BEGIN [WORK]
```

where:

BEGIN

Marks the beginning of one or more transactions.

WORK

An optional keyword with no effect.

Issuing the following command can disable autocommit mode for all sessions and threads:

```
SET AUTOCOMMIT=0
```

Once you have disabled autocommit, the *COMMIT* statement is required to store any and every data modification to disk, and the *ROLLBACK* statement is required to undo changes made during a transaction. Disabling autocommit is only effective with "transaction-safe tables" such as **InnoDB** or **BDB** tables. Disabling autocommit on non-transaction safe tables has no effect — autocommit will still be enabled.



Earlier versions of MySQL use an update log. However, the update log does not support ANSI transactions unless the tables are defined as InnoDB or BDB.

Transactions are stored in the binary log in a single write operation when the *COMMIT* statement is issued. Here's an example:

```
BEGIN;
```

```
SELECT @A := SUM(salary)
```

```
FROM employee
```

```
WHERE type=1;
```

```
UPDATE payhistory SET summary=@A WHERE type=1;
```

```
COMMIT;
```

Rollbacks issued against non-transactional tables fail with the error **ER_WARNING_NOT_COMPLETE_ROLLBACK**, but transaction safe tables will be restored as expected.

Oracle

Not supported. Transactions in Oracle are started implicitly. Refer to *SET TRANSACTION* on the Oracle platform for more information about how Oracle controls individual transactions.

PostgreSQL

The PostgreSQL syntax is:

BEGIN [WORK | TRANSACTION]

where:

BEGIN

Marks the beginning of one or more transactions.

WORK

An optional keyword with no effect.

TRANSACTION

An optional keyword with no effect.

PostgreSQL normally runs in autocommit mode where each data modification statement or query is its own transaction. PostgreSQL applies an implicit *COMMIT* for a transaction that completes without an error. PostgreSQL also automatically issues a *ROLLBACK* statement at the end of a statement that fails. The *BEGIN* statement allows explicit *COMMIT* or *ROLLBACK* of a transaction, which may then consist of multiple statements.

Manually coded transactions are much faster in PostgreSQL than are autocommitted transactions. The *SET TRANSACTION ISOLATION LEVEL* should be set to *SERIALIZABLE* just after the *BEGIN* statement to bolster the transaction isolation. PostgreSQL allows many data-modification statements (*INSERT*, *UPDATE*, *DELETE*) within a *BEGIN* . . . *COMMIT* block. However, when the *COMMIT* command is issued, either all or none of the transaction takes place, depending on the success or failure of the command.



BEGIN has a separate usage on database platforms that support their own procedural language, namely Oracle and SQL Server. In these platforms, *BEGIN*, without the keyword *TRANSACTION*, is used to mark a new block of procedural code. For this reason, you are advised to include the *TRANSACTION* keyword for any transactions you write on PostgreSQL. Otherwise, you will face some complicated migration issues, should you ever move your code to Oracle or SQL Server.

Following is an example of *BEGIN TRANSACTION* in PostgreSQL:

```
BEGIN TRANSACTION;
```

```
    INSERT INTO jobs(job_id, job_desc, min_lvl, max_lvl)
```

```
    VALUES(15, 'Chief Operating Officer', 185, 135)
```

```
COMMIT;
```

SQL Server

Microsoft SQL Server supports the *BEGIN TRANSACTION* statement rather than the ANSI *START TRANSACTION* statement. It also supports a couple of extensions that facilitate transaction backup and recovery. The Microsoft SQL Server syntax is:

```
BEGIN TRAN[SACTION] [transaction_descriptor
```

```
    [WITH MARK [ 'log_descriptor' ] ] ]
```

where:

TRAN[SACTION]

Marks the beginning of a transaction. SQL Server allows either *TRAN* or *TRANSACTION*.

transaction_descriptor

An up to 32-character name used to identify a transaction, or it can be a variable string datatype (*CHAR*, *NCHAR*, *VARCHAR*, or *NVARCHAR*) variable up to 32 characters in size. When working with nested transactions, only name the outermost transaction.

WITH MARK *log_descriptor*

Tells SQL Server to place a mark of name *log_descriptor* in the transaction log. This allows SQL Server to restore a transaction log up to the specified *log_descriptor*. In a sense, this allows point in time recovery based on the name of the mark for database set to *FULL* recovery mode. *WITH MARK* must be used in conjunction with a named transaction.

When nesting transactions, only the outermost *BEGIN . . . COMMIT* or *BEGIN . . . ROLLBACK* pair should reference the transaction name (if it has one). In general, we recommend avoiding nested transactions.

Here is a SQL Server set of *INSERT* statements, all performed as a single transaction:

BEGIN TRANSACTION

```
INSERT INTO sales VALUES('7896','JR3435','Oct 28 2003',25,  
'Net 60','BU7832')
```

```
INSERT INTO sales VALUES('7901','JR3435','Oct 28 2003',17,  
'Net 30','BU7832')
```

```
INSERT INTO sales VALUES('7907','JR3435','Oct 28 2003',6,  
'Net 15','BU7832')
```

COMMIT

GO

If, for any reason, any one of these *INSERT* statements had to wait for completion, they would all have to wait since they are treated as a single transaction.

See Also

COMMIT
ROLLBACK

SUBQUERY Substatement

A subquery is a nested query. Subqueries may appear in various places within a SQL statement.

Platform	Command
DB2	Supported
MySQL	Supported with limitations
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL supports the following types of subquery:

Scalar subqueries

Subqueries that retrieve a single value. These are the most widely supported type of subquery among the various database platforms.

Table subqueries

Subqueries that retrieve more than one values. This type of subquery retrieves more than a row of values.

Nested table subqueries

Subqueries that retrieve more than one column and more than one row.

Scalar and vector subqueries can, on some platforms, appear as part of the expression in a *SELECT* list of items, *WHERE* clause, and *HAVING* clause. Nested table subqueries tend to appear in the *FROM* clause of *SELECT* statements.

A *correlated subquery* is a subquery that is dependent upon a value in an outer query. Consequently, the inner query is executed once for every record retrieved in the outer query. Since subqueries can be nested many layers deep, a correlated subquery may reference any level in the main query higher than its own level.

Different rules govern the behavior of a subquery depending on which clause it appears in. The level of support amongst the database platforms also varies. Some platforms support subqueries in all clauses mentioned earlier (*SELECT*, *FROM*, *WHERE* and *HAVING*), while others support subqueries in only one or two of the clauses.

Subqueries are usually associated with the *SELECT* statement. Since subqueries may appear in the *WHERE* clause, they can be used in any SQL statement that supports a *WHERE* clause, including *SELECT*, *INSERT...SELECT*, *DELETE*, and *UPDATE* statements.

SQL2003 Syntax

Scalar, table, and nested table subqueries are represented in the generalized syntax below:

SELECT column1, column2, ... (scalar subquery)

FROM table1, ... (nested table subquery)

AS subquery_table_name]

WHERE foo = (scalar subquery)

OR foo IN (table subquery)

Correlated subqueries are more complex because the value of such subqueries is dependent on values retrieved in their main queries. For example:

SELECT column1

FROM table1 AS t1

WHERE foo IN

(SELECT value1

FROM table2 AS t2

WHERE t2.pk_identifier = t1.fk_identifier)

Note that the *IN* clause is for example purposes only. Any comparison operator may be used).

Keywords

scalar subquery

Includes a scalar subquery in the *SELECT* item list or in the *WHERE* or *HAVING* clause of a query.

nested table subquery

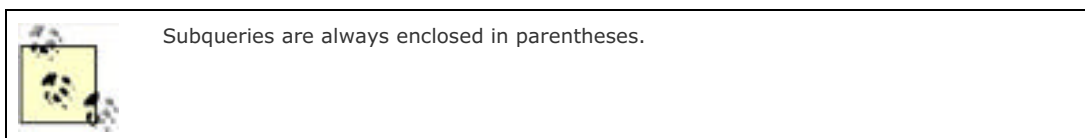
Includes a nested table subquery only in the *FROM* clause in conjunction with the *AS* clause.

table subquery

Includes a table subquery only in the *WHERE* clause with operators such as *IN*, *ANY*, *SOME*, *EXISTS*, or *ALL* that act upon multiple values. Table subqueries return one or more rows containing a single value.

Rules at a Glance

Subqueries allow you to return one or more values and nest them inside a *SELECT*, *INSERT*, *UPDATE*, or *DELETE* statement, or inside another subquery. Subqueries can be used wherever expressions are allowed. Subqueries also can often be replaced with a *JOIN* statement. Depending on the DBMS, subqueries may perform less quickly than joins.



Subqueries may appear in a *SELECT* clause with an item list containing at least one item, a *FROM* clause for referencing one or more valid tables or views, and in *WHERE* and *HAVING* clauses.

Scalar subqueries can return only a single value. Certain operators in a *WHERE* clause expect only one value, such as *=*, *<*, *>*, *>=*, *<=*, and *<>* (or *!=*). If a subquery returns more than one value against an operator that expects a single value, the entire query will fail. On the other hand, table subqueries may return multiple values, but are usable only with multivalued expressions like *[NOT] IN*, *ANY*, *ALL*, *SOME*, or *[NOT] EXISTS*.

Nested table subqueries may appear only in the *FROM* clause and should be aliased by the *AS* clause. The result set returned by the nested table subquery, sometimes called a *derived table*, offers similar functionality to a view (see *CREATE VIEW*). Every column returned in the derived table need not be used in the query, though they can all be acted upon by the outer query.

Correlated subqueries typically appear as a component of a *WHERE* or *HAVING* clause in the outer query (and, less commonly, in the *SELECT* item list) and are correlated through the *WHERE* clause of the inner query (that is, the subquery). (Correlated subqueries can also be used as nested table subqueries, though this is less common.) Be sure to include a *WHERE* clause in such a subquery that evaluates based on a correlating value from the outer query. (The example for a correlated query in the earlier ANSI syntax diagram illustrates this requirement.) It is important to specify a table alias, called a *correlation name*, using the *AS* clause or other alias shortcut for every table referenced in a correlated query, both in the outer and inner query. Correlation names avoid ambiguity and help the DBMS quickly resolve the tables involved in the query.

All ANSI-compliant subqueries comply with the following short list of rules:

- A subquery cannot include an *ORDER BY* clause.
- A subquery cannot be enclosed in an aggregate function. The following query is invalid: *SELECT foo FROM table1 WHERE sales >= AVG(SELECT column1 FROM sales_table...)*. You can get around this limitation by performing the aggregation in the subquery rather than in the outer query.

Programming Tips and Gotchas

For most all vendor platforms, subqueries should not reference large object datatypes (e.g., *CLOB* or *BLOB* on Oracle and *IMAGE* or *TEXT* on SQL Server), or array datatypes (such as *TABLE* or *CURSOR* on SQL Server).

The platforms all support subqueries, but not every vendor supports every type of subquery. [Table 3-6](#) tells you whether your platform supports each type of subquery.

Table 3-6. Platform-specific subquery support

Platform	DB2	MySQL	Oracle	PostgreSQL	SQL Server
Scalar subquery in SELECT item list	Y	Y	Y	X	Y
Scalar subquery in WHERE/HAVING clause	Y	Y	Y	Y	Y
Vector subquery in WHERE/HAVING clause	Y	Y	Y	Y	Y
Nested table in FROM clause	Y	Y	Y	Y	Y
Correlated subquery in WHERE/HAVING clause	Y	Y	Y	Y	Y

Subqueries are not relegated to *SELECT* statements only. They may also be used in *INSERT*, *UPDATE*, and *DELETE* statements that include a *WHERE* clause. Subqueries are often used for the following purposes:

- To identify the rows inserted into the target table using an *INSERT...SELECT* statement, *CREATE TABLE...SELECT* statement, or *SELECT...INTO*.
- To identify the rows of a view or materialized view in a *CREATE VIEW* statement.
- To identify value(s) assigned to existing rows using an *UPDATE* statement.
- To identify values for conditions in *WHERE* and *HAVING* clauses of *SELECT*, *UPDATE*, and *DELETE* statements.
- To build a view of a table(s) on the fly (i.e., nested table subqueries).

Examples

This section shows subquery examples that are equally valid on DB2, MySQL, Oracle, PostgreSQL, and SQL Server.

A simple scalar subquery is shown in the *SELECT* item list of the following query:

```
SELECT job, (SELECT AVG(salary) FROM employee) AS "Avg Sal"  
FROM employee
```

Nested table subqueries are functionally equivalent to querying a view. In the following, we query the education level and salary in a nested table subquery, and then perform aggregations on the values in the derived table in the outer query.

```
SELECT AVG(edlevel), AVG(salary)  
FROM (SELECT edlevel, salary  
      FROM employee) AS emprand  
GROUP BY edlevel
```



Remember that this query may fail, depending on the platform, without the *AS* clause to associate a name with the derived table.

The following query shows a standard table subquery in the *WHERE* clause expression. In this case, we want all project numbers for employees in the department 'A00':

```
SELECT projno  
FROM emp_act  
WHERE empno IN  
      (SELECT empno  
      FROM employee  
      WHERE workdept = 'A00')
```

The above subquery is executed only once for the outer query.

In the next example, we want to know the names of employees and their level of seniority. We get this result set through a correlated subquery:

```
SELECT firstname, lastname,  
       (SELECT COUNT(*))  
FROM employee senior  
WHERE employee.hiredate > senior.hiredate) as senioritytype  
FROM employee
```

Unlike the previous subquery, this subquery is executed one time for every row retrieved by the outer query. In a query like this, the total processing time could be very long, since the inner query may potentially execute many times for a single result set.

Correlated subqueries depend on values retrieved by the outer query before being able to complete the processing of the inner query. They are tricky to master, but offer unique programmatic capabilities. The following example returns information about orders where the quantity sold in each order is less than the average quantity in other sales for that title:

```
SELECT s1.ord_num, s1.title_id, s1.qty  
FROM sales AS s1  
WHERE s1.qty <  
       (SELECT AVG(s2.qty)  
        FROM sales AS s2  
        WHERE s2.title_id = s1.title_id)
```

For this example, you can accomplish the same functionality using a self join. However, there are situations in which a correlated subquery may be the only easy way to do what you need.

The next example shows how a correlated subquery might be used to update values in a table:

```
UPDATE course SET ends =  
       (SELECT min(c.begins) FROM course AS c  
        WHERE c.begins BETWEEN course.begins AND course.ends)  
WHERE EXISTS  
       (SELECT * FROM course AS c  
        WHERE c.begins BETWEEN course.begins AND course.ends)
```

Similarly, you can use a subquery to determine which rows to delete. This example uses a correlated subquery to delete rows from one table based on related rows in another table:

```
DELETE FROM course  
WHERE EXISTS  
       (SELECT * FROM course AS c  
        WHERE course.id > c.id  
        AND (course.begins BETWEEN c.begins  
            AND c.ends OR course.ends BETWEEN c.begins AND c.ends))
```

DB2

DB2 supports the ANSI standard subquery types. It allows scalar subqueries in the *SELECT* item list, nested table subqueries in the *FROM* clause, and scalar and vector subqueries in the expressions of *WHERE* and *HAVING* clauses.

DB2 allows correlated subqueries in the *SELECT* item list and in *WHERE* and *HAVING* clauses.

MySQL

MySQL supports nested table subqueries as select items and in the *WHERE* clause.

Oracle

Oracle supports ANSI standard subqueries, though it uses a different nomenclature. In Oracle, a nested table subquery that appears in the *FROM* clause is called an *inline view*. That makes plenty of sense because nested table subqueries are basically views built on the fly. Oracle calls a subquery that appears in the *WHERE* clause and *HAVING* clause of a query a *nested subquery*. Oracle allows correlated subqueries in the *SELECT* item list and in the *WHERE* and *HAVING* clauses.

PostgreSQL

PostgreSQL supports ANSI standard subqueries in the *FROM*, *WHERE*, and *HAVING* clauses. However, subqueries appearing in a *HAVING* clause cannot include *ORDER BY*, *FOR UPDATE*, and *LIMIT* clauses. PostgreSQL does not currently support subqueries in the *SELECT* item list.

SQL Server

SQL Server supports ANSI standard subqueries. Scalar subqueries can be used almost anywhere a standard expression is allowed. Subqueries in SQL Server cannot include the *COMPUTE* or *FOR BROWSE* clauses. They can include the *ORDER BY* clause if the *TOP* clause is also used.

See Also

DELETE
INSERT
SELECT
UPDATE
WHERE

TRUNCATE TABLE Statement

The *TRUNCATE TABLE* statement, a non-ANSI statement, irrevocably removes all rows from a table without logging the individual row deletes. It quickly erases all the records in a table without altering the table structure, taking little or no space in the redo logs or transaction logs. However, since a truncate operation is not logged, the *TRUNCATE TABLE* statement cannot be rolled back once it is issued.

Platform	Command
DB2	Not supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

Defacto Standard Syntax

Officially, *TRUNCATE TABLE* is not an ANSI standard command. However, it is a commonly supported statement standard that follows this format:

TRUNCATE TABLE *table_name*

Keywords

table_name

The name of any valid table within the current database or schema context.


Rules at a Glance

The *TRUNCATE TABLE* statement has the same effect on a table as a *DELETE* statement with no *WHERE* clause; both erase all rows in a given table. However, there are two important differences. *TRUNCATE TABLE* is faster and it is non-logged, meaning it cannot be rolled back if issued in error. Plus, *TRUNCATE TABLE* does not activate triggers, while the *DELETE* statement does.

This command should be issued manually. We strongly encourage you not to place it into automated scripts or production systems that contain irreplaceable data. It cannot be paired with transaction control statements such as *BEGIN TRAN* or *COMMIT*.

Programming Tips and Gotchas

Because the *TRUNCATE TABLE* statement is not logged, it is generally used only in development databases. Use it in production databases with caution!

	We strongly advise that you do not write <i>TRUNCATE TABLE</i> statements into the stored procedures or functions of a production application because most platforms do not log the operation nor can they recover an improperly issued <i>TRUNCATE</i> statement.
---	--

The *TRUNCATE TABLE* command will fail if another user has a lock on the table at the time the statement is issued. *TRUNCATE TABLE* does not activate triggers, but it will work when they are present. However, it won't work when foreign key constraints are in place on a given table.

DB2

As of Version 8, DB2 does not support the *TRUNCATE TABLE* statement.

MySQL

MySQL supports a basic format for the *TRUNCATE TABLE* statement starting in Version 3.23:

```
TRUNCATE TABLE name
```

MySQL achieves the results of a *TRUNCATE TABLE* command by dropping and recreating the affected table. Since MySQL stores each table in a file called *table_name.frm*, the *table_name.frm* file must exist in the directory containing the database files for the command to work properly.

For example, to remove all data from the **publishers** table:

```
TRUNCATE TABLE publishers
```

Oracle

Oracle allows a table or an indexed cluster (but not a hash cluster) to be truncated. Oracle's syntax is:

```
TRUNCATE { CLUSTER [owner.]cluster  
| TABLE [owner.]table [{PRESERVE | PURGE} MATERIALIZED VIEW LOG]}  
[{{DROP | REUSE} STORAGE}]
```

TRUNCATE TABLE was first introduced by Oracle, and other platforms soon added support for the statement. Oracle has added more features to this statement than are commonly implemented by other vendors. Oracle's syntax elements are as follows:

CLUSTER | TABLE

Specifies whether an index cluster or a table will be truncated. An index cluster is a physical construct that stores related rows of two tables next to each other physically on disk to speed up joins by lowering I/O. *SNAPSHOT LOG* and *STORAGE* options are not available when truncating an index cluster.

PRESERVE MATERIALIZED VIEW LOG

Maintains any snapshot logs when a master table is truncated.

PURGE MATERIALIZED VIEW LOG

Clears out any snapshot logs.

DROP STORAGE

Causes the disk space freed by the deleted rows to be deallocated.

REUSE STORAGE

Causes the space allocated to a table to remain allocated to that table, though empty.

For example:

```
TRUNCATE TABLE scott.authors
    PRESERVE SNAPSHOT LOG REUSE STORAGE
```

This example command erases all records in the table **scott.authors**. It also maintains the existing snapshot log and allows the table to keep and reuse the storage space already allocated to it.

PostgreSQL

PostgreSQL varies little from the defacto industry standard, but does add the optional *TABLE* keyword:

```
TRUNCATE [TABLE] name
```

The *TABLE* component of the command is optional. For example, the following command erases all the records in the **authors** table on a PostgreSQL database:

```
TRUNCATE authors
```

SQL Server

SQL Server supports the defacto industry standard.

See Also

DELETE

UNION Set Operator

The *UNION* set operator combines the result sets of two or more queries, showing all the rows returned by each of the queries as one, single result set.

UNION is in a class of keyword called a *set operator*. Other set operators include *INTERSECT* and *EXCEPT/MINUS*.

(*EXCEPT* and *MINUS* are functionally equivalent. *EXCEPT* is the ANSI standard.) All set operators are used to simultaneously manipulate the result sets of two or more queries; hence the term "set operators."

Platform	Command
DB2	Supported, with limitations
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

There are technically no limits to the number of queries that you may combine with the *UNION* statement. The general syntax is:

<SELECT *statement1*>

UNION [ALL | DISTINCT]

<SELECT *statement2*>

UNION [ALL | DISTINCT]

...

Keywords

UNION

Determines which result sets will be combined into a single result set. Duplicate rows are, by default, excluded.

ALL

Combines duplicate rows from all result sets.

DISTINCT

Eliminates duplicate rows from the final result set. Columns containing a NULL value are considered duplicates. (If neither *ALL* nor *DISTINCT* are used, *DISTINCT* behavior is the default.)

Rules at a Glance

There is only one significant rule to remember when using *UNION*: the order, number, and datatype of columns should be the same in all queries.

The datatypes do not have to be identical, but they should be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest of two (or more) compatible datatypes, and a query that unions three *CHAR* columns (*CHAR(5)*, *CHAR(10)*, and *CHAR(12)*) will display the results in the *CHAR(12)* format with extra space padded onto the smaller column results.

Programming Tips and Gotchas

Even though the ANSI standard calls for *INTERSECT* to take precedence over other set operators in a single statement, many platforms evaluate all set operators with equal precedence. Explicitly control the precedence of set operators using parentheses. Otherwise, the DBMS is likely to evaluate them from leftmost to rightmost expression.

Depending on the platform, *DISTINCT* can incur a significant performance cost, since it often adds a second pass through the results to winnow out duplicate records. *ALL* can be specified in any instance where no duplicate records are expected (or where duplicate records are OK) for faster results.

According to the ANSI standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. In order to avoid column and table ambiguity, be sure to alias each column for each table with the same respective alias. However, for column naming purposes, only the alias in the first query is used to each column in the *SELECT...UNION* query. For example:

```
SELECT au_lname AS "lastname", au_fname AS "firstname"

FROM authors

UNION

SELECT emp_lname AS "lastname", emp_fname AS "firstname"

FROM employees

ORDER BY lastname, firstname
```

Also, while your query may have compatible datatyped columns throughout the queries in the *UNION*, there may be some variation in behavior across the DBMS platforms, especially with regard to varying length of the columns. For example, if the **au_lname** column in the first query is markedly longer than the **emp_lname** column in the second query, each platform may apply different rules as to which length is used. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the columns across the tables have different names. In general, the column names of the first query are used.

DB2

DB2 supports the *UNION* and *UNION ALL* keywords of the ANSI standard, plus the *VALUES* clause:

```
<SELECT statement1>

UNION [ALL]

<SELECT statement2>

UNION [ALL]

...

[VALUES (expression1, expression2,
...)] [, (expression1, expression2, ...)] [,...]
```

where:

VALUES

Allows you to specify one or more sets of hand-coded values as records in the combined result set. Each set of hand-coded values must contain the exact same number of columns as the queries in the *UNION*. Separate rows of hand-coded values in the record set using commas.

Although *UNION DISTINCT* is not supported, *UNION* is the functional equivalent. The *CORRESPONDING* clause is not supported.

The datatypes *LONG VARCHAR*, *LONG VARGRAPHIC*, *BLOB*, *CLOB*, *DBCLOB*, *DATALINK*, and structured types are not usable in a *UNION* (but are usable in *UNION ALL*).

If the result set has the same column name across tables, that name is used. If, however, the tables have different column names, DB2 generates a new column name. That column is then unusable in the *ORDER BY* clause or the *FOR UPDATE* clause.

If multiple set operations are used in a single query, those enclosed in parentheses are evaluated first. Then, the queries are evaluated in left-to-right order. However, all *INTERSECT* operations are performed before any *UNION* and *EXCEPT* operations. For example:

```
SELECT empno
FROM employee
WHERE workdept LIKE 'E%'
UNION
SELECT empno
FROM emp_act
WHERE projno IN ('IF1000', 'IF2000', 'AD3110')
UNION
VALUES ('AA0001'), ('AB0002'), ('AC0003')
```

The example above retrieves all employee IDs from the employee table that are in any department that starts with "E," as well as any employee IDs from the employee accounts (**emp_act**) table working in the projects **IF1000**, **IF2000**, and **AD3110**, and always includes employee IDs **AA0001**, **AB0002**, and **AC0003**.

MySQL

Not supported

Oracle

Oracle supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
UNION [ALL]
<SELECT statement2>
UNION [ALL]
...
```

Oracle does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent. Oracle does not support *UNION* or *UNION ALL* on queries under the following circumstances:

- Queries containing columns of *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatype.
- Queries containing a *FOR UPDATE* clause or a *TABLE* collection expression.

If the first query in the set operation contains any expressions in the select item list, then include the *AS* statement to associate an alias with the column. Also, only the last query in the set operation may contain an *ORDER BY* clause. For example, you could find out all unique store IDs without duplicates using this query:

```
SELECT stor_id FROM stores
UNION
SELECT stor_id FROM sales;
```

PostgreSQL

PostgreSQL supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
```

UNION [ALL]

<SELECT *statement2*>

UNION [ALL]

...

PostgreSQL does not support *UNION* or *UNION ALL* on queries with a *FOR UPDATE* clause. PostgreSQL does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *UNION* or *UNION ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the rightmost occurrence of *ORDER BY* or *LIMIT* will be assumed to apply to the entire set operation.

For example, we could find all authors and all employees whose last name starts with "P":

```
SELECT a.au_lname
```

```
FROM authors AS a
```

```
WHERE a.au_lname LIKE 'P%'
```

```
UNION
```

```
SELECT e.lname
```

```
FROM employee AS e
```

```
WHERE e.lname LIKE 'W%';
```

SQL Server

SQL Server supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

<SELECT *statement1*>

UNION [ALL]

<SELECT *statement2*>

UNION [ALL]

...

SQL Server does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent.

You can use *SELECT...INTO* with *UNION* or *UNION ALL*, but *INTO* may only appear in the first query of the union. Special keywords, such as *SELECT TOP* and *GROUP BY...WITH CUBE*, are usable with all queries in a union. However, be sure to include them with all queries in the union. If you use *SELECT TOP* or *GROUP BY...WITH CUBE* in only one query, the operation will fail.

Each query in a union must contain the same number of columns. The datatypes of the columns do not have to be identical, but they must implicitly convert. For example, mixing *VARCHAR* and *CHAR* columns is acceptable. When returning data, SQL Server uses the larger of the two columns when evaluating the size of the columns returned in the result set. Thus, if a *SELECT...UNION* statement has a *CHAR(5)* column and a *CHAR(10)* column, it will display the data of both columns as a *CHAR(10)* column. Numeric columns are converted to and displayed as the most precise datatype in the union.

For example, the following query unions the results of two independent queries that use *GROUP BY...WITH CUBE*:

```
SELECT ta.au_id, COUNT(ta.au_id)
```

```
FROM pubs..titleauthor AS ta
```

```
JOIN pubs..authors AS a ON a.au_id = ta.au_id
```

```
WHERE ta.au_id >= '722-51-5454'
```

```
GROUP BY ta.au_id WITH CUBE
UNION
SELECT ta.au_id, COUNT(ta.au_id)
FROM pubs..titleauthor AS ta
JOIN pubs..authors AS a ON a.au_id = ta.au_id
WHERE ta.au_id < '722-51-5454'
GROUP BY ta.au_id WITH CUBE
```

See also

EXCEPT
INTERSECT
MINUS
SELECT

UPDATE Statement

The *UPDATE* statement changes existing data in a table. Use great caution when issuing an *UPDATE* statement without a *WHERE* clause, since the statement then affects every row in the entire table.

Platform	Command
DB2	Supported, with variations
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
UPDATE [ONLY] {table_name | view_name}
SET {{column_name = { ARRAY [array_val [,...]] | DEFAULT |
NULL | scalar_expression},
column_name = { ARRAY | DEFAULT |
NULL | scalar_expression}
[,...]}
| ROW = row_expression }
[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

Keywords

ONLY

Restricts cascading of the updated values to any subtables of the target table or view. *ONLY* effects only typed (object-oriented) tables and views. If used with a nontyped table or view, it causes an error.

table_name | *view_name*

The target table or view of the update statement. You need appropriate permissions on the target according to the rules of the platform. Updates against views often have special rules. Generally, it is only advisable to perform an *UPDATE* against a view if the view is representative of a single table.

SET

Assigns a specific value to a column or row.

column_name

Used in conjunction with *SET*, as in *SET column1= 'foo'*. Allows you to set a new value for the specified column. You can update as many columns as you like in one statement, though you cannot update the same column more than once in a single statement.

ARRAY [*array_val* [,...]]

Assigns the column to an array value or to an empty array (using *ARRAY[]*). This behavior is not widely supported by DBMS platforms.

NULL

Sets the column to NULL.

DEFAULT

Sets the column to its default value as defined by a *DEFAULT* specification.

scalar_expression

Sets the column to any single value expression, such as a literal string or numeric value, a scalar function, or a scalar subquery.

ROW

Used in conjunction with *SET* is a mutually exclusive option to *SET column_name*, as in *SET ROW = ROW('foo','bar')*. This behavior is not widely supported by DBMS platforms.

row_expression

Used to set a value for each and every column in the table.

WHERE *search_condition*

Defines search criteria for the *UPDATE* statement using the *search_condition* to ensure that only the target rows are updated. Any legal *WHERE* clause is acceptable. Typically, these criteria are evaluated against each row of the table before the update is applied. If the search criteria is a subquery, the subquery is executed on each row — potentially a long-running process.

WHERE CURRENT OF *cursor_name*

Restricts the *UPDATE* statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

The *UPDATE* statement can be used to modify the value of one or more columns of a table or view at a time. Typically,

you will update the values in one or more rows at a time. The new values must be scalar (except in row expressions and arrays). That is, the new value must have a single, constant value at the time the transaction is executed, though it can be literal or derived from a function call or subquery.

A basic *UPDATE* statement without a *WHERE* clause looks like this:

```
UPDATE authors
SET contract = 0
```

Without a *WHERE* clause, all authors in the **authors** table have their contract status set to zero (meaning they don't have a contract any more). Similarly, values can be adjusted mathematically with an *UPDATE* statement:

```
UPDATE titles
SET price = price * 1.1
```

This *UPDATE* statement would increase all book prices by 10%.

You can set multiple values in one statement as here, where we set all author last name and first names to uppercase letters:

```
UPDATE authors SET au_lname = UPPER(au_lname),
    au_fname = UPPER(au_fname);
```

Adding a *WHERE* clause to an *UPDATE* statement allows records in the table to be modified selectively:

```
UPDATE titles
SET type = 'pers_comp',
    price = (price * 1.15)
WHERE type = 'popular_com';
```

The above query makes two changes to any record of the type '**popular_com**'. The command increases their price by 15% and alters their type to '**pers_comp**'.

You can also invoke functions or subqueries to derive the values used in an *UPDATE* statement. In some cases, you may wish to update the specific row that is being processed by a declared and open cursor. The following example shows both concepts:

```
UPDATE titles SET ytd_sales = (SELECT SUM(qty)
    FROM sales
    WHERE title_id = 'TC7777')
WHERE CURRENT OF title_cursor;
```

This query assumes that you have declared and opened a cursor named **title_cursor** and that it is processing titles with an ID of **TC7777**.

Sometimes you need to update values in a given table based on the values stored in another table. For example, if you need to update the publication date for all the titles written by a certain author, you might find the author and list of titles first through subqueries:

```
UPDATE titles
SET pubdate = 'Jan 01 2002'
WHERE title_id IN
    (SELECT title_id
    FROM titleauthor
```

```
WHERE au_id IN  
  
(SELECT au_id  
  
FROM authors  
  
WHERE au_lname = 'White'))
```

Programming Tips and Gotchas

The *UPDATE* statement alters values in existing records of a table or view. If you update a view, the view should include all necessary (*NOT NULL*) columns, or else the statement may fail. Columns with a *DEFAULT* value can usually be safely omitted.

The *SET ROW* clause is not widely supported among DBMS platforms. Avoid this clause. The *WHERE CURRENT OF* clause is more commonly implemented on DBMS platforms, but must be used in conjunction with a cursor. Make sure you understand the use and functionality of cursors before using this clause.

In an interactive user session, it is good practice to issue a *SELECT* statement using the same *WHERE* clause before issuing the actual *UPDATE* statement. This precaution enables you to check all rows in the result set before actually performing the *UPDATE*, helping ensure that you don't alter anything you don't mean to.

DB2

DB2 supports most of the standard structures found in the ANSI standard. It includes transaction control features using a transaction isolation level clause (the *WITH* clause), as well as typed table and view modification, but does *not* include the *SET ROW* clause.

```
UPDATE [ONLY] ( { table_name | view_name } ) AS alias  
  
SET { { column_name1 = { DEFAULT | NULL | expression },  
  
      column_name2 = { DEFAULT | NULL | expression }  
  
      [,... ] } |  
  
      { ( column_name1[..attribute_name],  
  
        column_name2[..attribute_name][,...] ) =  
  
        ( {DEFAULT | NULL | expression},  
  
          {DEFAULT | NULL | expression}[,...] ) |  
  
        SELECT statement }  
  
[WHERE {search_conditions | CURRENT OF cursor_name}]  
  
[WITH { RR | RS | CS | UR }]
```

where:

AS alias

Specifies an alias for the target table or view. This is useful for *UPDATE* statements that involve lengthy *WHERE* clauses or correlated subqueries.

*..*attribute_name**

Specifies the *attribute_name* when updating a column that is a structured type.

WITH

Controls the transaction isolation level for the update statement (*RR* is repeatable read, *RS* is read stability, *CS* is cursor stability, and *UR* is uncommitted read). See *SET TRANSACTION* for details on DB2's transaction isolation levels.

The *UPDATE* statement must have only one value assignment per column. A column value can be set only once in an *UPDATE* statement. If any values of the update violate a constraint on the table or otherwise cause an error, then none of the columns are updated. When updating the value of an identity column to the current identity value, just use *DEFAULT* for that column. (DB2 fully supports updates on structure types, but they are neither commonly used nor quick and easy to explain. Refer to the DB2 documentation for more details on updating structured types.) DB2 allows parenthetical *SET* clauses where multiple target columns and values are assessed at once (see example later).

DB2 not only allows updates against standard tables and views, but also against typed (object-oriented) tables and views. (Refer to *CREATE TABLE* and *CREATE TYPE* for more details on these.) There are some special rules for updating against typed tables and views:

- The object ID column is not updateable.
- *..attribute_name* must be used in conjunction with a *column_name* that was previously defined as a user-defined structured type. The *attribute_name* must be an attribute of the structured type identified as *column_name*.
- Updating the column distribution statistics of a typed table requires specification of the subtable that first introduced the column.
- Multiple attribute assignments on the same structured type column occur in left-to-right order for a *SET* clause in parentheses, or else in the order specified in the *SET* clause.
- The same column may appear many times in a single *UPDATE* statement as long as each appearance sets a value for a specific *attribute_name*.

Updating the URL value of *DATALINK* columns is the same as deleting and then re-inserting them. Updating a *DATALINK* column with NULL is the same as deleting it. Updating the comment value of a *DATALINK* without relinking requires passing an empty string for the URL value.

Here are two examples that show how DB2 updates multiple values at once using the standard *SET* clause:

```
UPDATE employee
SET job=NULL, salary=0, bonus=0, comm=0
WHERE workdept = 'E21'
AND job <> 'MANAGER'
```

This statement could also be written using a parenthetical *SET* clause, as follows:

```
UPDATE employee
SET (job, salary, bonus, comm) = (NULL, 0, 0, 0)
WHERE workdept = 'E21'
AND job <> 'MANAGER'
```

Here is an example of a DB2 update against a base table on a type called *circles*. The *circles* attributes are owner, radius, X-axis center, and Y-axis center. Since the column **C** is defined as a type of **circle**, the *UPDATE* statement is changing the attributes of a single column:

```
UPDATE CIRCLES
SET (OWNER, C..RADIUS_LENGTH, C..X_CENTER, C..Y_CENTER) =
('tony', 5, 18, 22)
WHERE ID = 15
```

MySQL

MySQL supports the ANSI standard with a few variations, including the *LOW PRIORITY* clause, the *IGNORE* clause, and the *LIMIT* clause:

```
UPDATE [LOW PRIORITY] [IGNORE] table_name
SET column_name = {scalar_expression}
    [,...]
WHERE search_conditions
[ORDER BY column_name1 [{ASC | DESC}] [,...]]
[LIMIT integer]
```

where:

LOW PRIORITY

Tells MySQL to delay the execution of the *UPDATE* statement until no other client is reading from the table.

IGNORE

Tells MySQL to ignore any duplicate key errors generated by *PRIMARY KEY* and *UNIQUE* constraints. However, only records that don't generate such errors will be updated.

ORDER BY

Tells MySQL to update the rows in the specified order (ascending or descending) of the columns specified.

LIMIT integer

Restricts the *UPDATE* action to a specific number of rows as designated by the integer value.

MySQL supports *DEFAULT* value assignment in the *INSERT* statement, but not currently in the *UPDATE* statement.

The following *UPDATE* would increase all prices in the **titles** table by 1:

```
UPDATE titles SET price = price + 1;
```

The following example limits the *UPDATE* to the first 10 records encountered in the **titles** table according to **title_id**. Also, the value of **price** is updated twice. Those two updates of **price** are assessed from left to right. First, the price is doubled, then it is increased by 1:

```
UPDATE titles
SET price = price * 2,
    price = price + 1
ORDER BY title_id
LIMIT 10;
```

Oracle

The Oracle implementation of *UPDATE* allows updates against views, materialized views, subqueries, and tables in an allowable schema:

```
UPDATE [ONLY]
{ [schema}.{view_name | materialized_view_name |
    table_name}
    [ @database_link ] [alias]
```

```
{[PARTITION (partition_name) |  
 [SUBPARTITION (subpartition_name)]}  
 | subquery [WITH {[READ ONLY] | [CHECK OPTION [CONSTRAINT constraint_name ]]  
 | [TABLE (collection_expression_name) [ ( + ) ] ] }  
SET {column_name1 [,...] = {expression [,...] | subquery} |  
  VALUE [(alias)] = { value | (subquery)},  
 {column_name1 [,...] = {expression [,...] | subquery} |  
  VALUE [(alias)] = { value | (subquery)},  
 [,...]  
WHERE search_conditions | CURRENT OF cursor_name}  
RETURNING expression [,...] INTO variable [,...];
```

Syntax elements for Oracle's version of *UPDATE* are:

ONLY

Tells Oracle not to update rows from any subviews. Applies only to views that belong to a hierarchy.

materialized_view_name

Applies the update statement to a pre-existing snapshot.

@database_link

Applies the update statement to a view or table on a remote server made available through a pre-existing database link

alias

Specifies an alias for the table being updated. Oracle allows table aliases, but does not support the *AS* keyword with *UPDATE* statements. Aliases can also be supplied for the *VALUE* clause of an *UPDATE* statement.

PARTITION

Applies the update to a named partition, rather than to an entire table. You are not required to name a partition when updating a partitioned table, but doing so can, in many cases, help reduce the complexity of your *WHERE* clause.

SUBPARTITION

Applies the update to a named subpartition, rather than to an entire table.

WITH READ ONLY

Specifies that the *subquery* cannot be updated. In some cases, an *UPDATE* transaction may actually alter the records used in a subquery. The *WITH READ ONLY* clause prevents this from happening.

WITH CHECK OPTION

Tells Oracle to abort any changes to the updated table that would not appear in the result set of the subquery.

CONSTRAINT

Tells Oracle to further restrict changes based upon a specific constraint.

SET VALUE

Allows you to set the entire row value for any *TABLE* datatype. The *SET VALUE* clause is very similar to the ANSI *SET ROW* clause. You can also specify a subquery that retrieves all values needed by the *SET VALUE* clause.

RETURNING

Retrieves the rows affected by the command, whereas *UPDATE* normally only shows the number of rows updated. When used for a single-row update, the values of the row can be stored in PL/SQL variables and bind variables. When used for a multirow delete, the values of the rows are stored in bind arrays.

INTO

Indicates the variables into which the values brought back by the *RETURNING* clause should be stored.

Oracle has some important rules about issuing *UPDATE* statements:

- *UPDATE* statements are *not* allowed against tables or the base tables of views containing a domain index with a status of *FAILED* or *IN PROGRESS*. *UPDATE* statements are also not allowed against any table having an index partition with a status of *UNUSABLE*. (You can avoid the index partition problem by setting the *SKIP_UNUSABLE_INDEXES* session parameter to *TRUE*.)
- Updates against views can be problematic if a view's defining query contains a join, an aggregate function, a set operator, the *DISTINCT* keyword, an *ORDER BY* clause, a *GROUP BY* clause, a *CONNECT BY* clause, a *START WITH* clause, an analytical function, a collection expression, or a subquery. (Under these circumstances, you can use an *INSTEAD OF* trigger to update the tables underlying a view.)

The following code snippets show some examples of the extensions that Oracle offers in the *UPDATE* statement. To begin with, assume that the **sales** table has grown into a large partitioned table. You can update a specific partition using the *PARTITION* clause, for example:

```
UPDATE sales PARTITION (sales_yr_2004) s
```

```
    SET s.payterms = 'Net 60'
```

```
WHERE qty > 100;
```

You can also use the *SET VALUE* clause to supply a new value for each column in the row identified by the *WHERE* clause:

```
UPDATE big_sales bs
```

```
    SET VALUE(bs) = (SELECT VALUE(s) FROM sales s
```

```
        WHERE bs.title_id = s.title_id)
```

```
        AND bs.stor_id = s.stor_id)
```

```
WHERE bs.title_id = 'TC777';
```

You can return values for review after an update by using the *RETURNING* clause. The following example updates one row, placing the updated values into PL/SQL variables:

```
UPDATE employee
```

```
    SET job_id = 13, job_level = 140
```

```
WHERE last_name = 'Josephs'
```

```
RETURNING last_name, job_id
```

```
INTO :var1, :var2;
```

PostgreSQL

PostgreSQL supports the ANSI standard *UPDATE* syntax with a couple of small variations. PostgreSQL supports the addition of a *FROM* clause. PostgreSQL's other variations include no support for the *ARRAY* keyword, but support for an array functionality. PostgreSQL's *UPDATE* syntax is:

```
UPDATE [ONLY] {table_name | view_name}

SET column_name1 = {DEFAULT | NULL | scalar_expression },
    column_name2 = {DEFAULT | NULL | scalar_expression }
    [,...]

[FROM {table1 [AS alias], table2 [AS alias] [,...]}]

[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

Most syntax elements are the same as in the ANSI standard. The only non-standard element is:

FROM

Provides the ability to build highly selective join-based criteria when determining which rows to update. *FROM* is not needed when only one table, the target table, is used to determine the rows updated.

In the following example, we want to update **job_lvl** for employees who have a **job_id** of 12 and a **min_lvl** of 25. We could do this with a large, complex subquery, or we could use the *FROM* clause to enable us to build a join:

```
UPDATE employee

SET job_lvl = 80

FROM employee AS e, jobs AS j

WHERE e.job_id = j.job_id

AND j.job_id = 12

AND j.min_lvl = 25;
```

All other aspects of the ANSI standard are fully supported.

SQL Server

SQL Server supports most of the basic components of the ANSI *UPDATE* statement, but it does not support the *ONLY* and *ARRAY* keywords, nor does it support array update functionality. SQL Server has extended the capabilities of *UPDATE* by adding table hints using the *WITH* clause, query hints using the *OPTION* clause, as well as more robust variable handling, as follows:

```
UPDATE {table_name | view_name | rowset}

[WITH (hint1, hint2 [,...])]

SET {column_name = {DEFAULT | NULL | scalar_expression }
    | variable_name = scalar_expression
    | variable_name = column_name = scalar_expression } [,...]

[FROM {table1 | view1 | nested_table1 | rowset1} [,...]]

[AS alias]

[JOIN {table2 [,...]}]

WHERE {conditions | CURRENT OF [GLOBAL] cursor_name}
```

[OPTION (*hint1*, *hint2* [,...])]

SQL Server's *UPDATE* syntax elements are as follows:

WITH *hint*

Allows the use of table hints to override the default behavior of the query optimizer. Since the query optimizer is quite good at choosing query plans, use hints only with a deep understanding of the tables, indexes, and data affected by the operation. Without this understanding, including hints could result in a decrease rather than an increase in performance.

variable_name

SQL Server variables must be declared prior to the *UPDATE* statement in the form of *DECLARE @variable*. The construct *SET @variable = column1 = expression1* sets a variable to the final value of an updated column, whereas *SET @variable = column1, column1 = expression* sets the variable to the value of the column before execution of the *UPDATE* statement.

FROM

Provides the ability to build highly selective join-based criteria when determining which rows to update. *FROM* is not needed when only one table, the target table, is used to determine the rows updated. SQL Server rowset functions are described in the [SELECT Statement](#) section.

AS *alias*

Allows you to assign an easy-to-use alias to the table, view, nested table subquery, or rowset function.

JOIN

Provides the ability to use ANSI standard syntax for joined tables, in conjunction with the *FROM* clause.

GLOBAL

A slight variation on the ANSI standard *WHERE CURRENT OF* clause. The clause *WHERE CURRENT OF cursor_name*, when used in combination with a cursor, tells SQL Server to update only the single record where the cursor is currently positioned. The cursor is assumed to be a local cursor, but can be designated a global cursor by using the keyword *GLOBAL*.

OPTION *hint*

Allows the use of query hints to override the default behavior of the query optimizer. Like the *WITH* clause, which uses table hints, use query hints only with a deep understanding of the tables, indexes, and data affected by the operation. Without this understanding, including hints could result in a decrease rather than an increase in performance.

The primary extension to the ANSI standard that Microsoft SQL Server offers in an *UPDATE* statement is a *FROM* clause. This *FROM* clause allows the use of the *JOIN* statement to make it especially easy to update rows in the target table by correlating rows declared in the *FROM* clause to the rows updated by the *UPDATE table_name* component of the statement. The following example shows an update using the ANSI style and a rather cumbersome subquery, followed by an update using SQL Server's *FROM* clause extension, to update the result of a table join. Both statements accomplish the same work, but in very different ways:

-- ANSI style

UPDATE titles

SET pubdate = GETDATE()

WHERE title_id IN

(SELECT title_id

FROM titleauthor


```
WHERE au_id IN
  (SELECT au_id
   FROM authors
   WHERE au_lname = 'White'))
-- Microsoft Transact-SQL style
UPDATE titles
SET pubdate = GETDATE( )
FROM authors AS a
JOIN titleauthor AS t2 ON a.au_id = t2.au_id
WHERE t2.title_id = titles.title_id
AND a.au_lname = 'White'
```

Performing this update using the Transact-SQL style is simply a matter of joining two tables — **authors** and **titleauthor** — to the **titles** table. To perform the same operation using ANSI-compliant code, the **au_id** in **author** must first be found and passed up to the **titleauthors** table, and then the **title_id** must be identified and passed up to the main update statement.

The following example updates the **state** column for the first 10 authors from the **authors** table:

```
UPDATE authors
SET state = 'ZZ'
FROM (SELECT TOP 10 * FROM authors ORDER BY au_lname) AS t1
WHERE authors.au_id = t1.au_id
```

The important thing to note about this example is that it is normally difficult to update the first *n* records in an *UPDATE* statement unless there is some explicit row sequence you can identify in the *WHERE* clause. However, the nested table subquery in the *FROM* clause uses a *TOP* keyword to return on the first 10 records, thereby saving a lot of added programming that would otherwise be required.

See Also

```
DECLARE
SELECT
WHERE
```

WHERE Clause

The *WHERE* clause sets the search criteria for an operation such as *SELECT*, *UPDATE*, or *DELETE*. Any records in the target table(s) that do not meet the search criteria are excluded from the operation. The search conditions may include many variations such as calculations, Boolean operators, and SQL predicates (for example, *LIKE* or *BETWEEN*).

Platform	Command
DB2	Supported
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

{ WHERE *search_criteria* | WHERE CURRENT OF *cursor_name* }

Keywords

WHERE *search_criteria*

Defines search criteria for the statement to ensure that only the target rows are affected.

WHERE CURRENT OF *cursor_name*

Restricts the operation of the statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

WHERE clauses are found in *SELECT* statements, *DELETE* statements, *INSERT...SELECT* statements, *UPDATE* statements, and any statement that might have a query or subquery such as *DECLARE*, *CREATE TABLE*, *CREATE VIEW*, and so forth. The search conditions, all of which are described in their own entries elsewhere in the book, can include:

All records (=ALL, >ALL, <= ALL, SOME/ANY)

For example, to see publishers who live in the same city as their authors:

```
SELECT pub_name
FROM publishers
WHERE city = SOME (SELECT city FROM authors);
```

Combinations (AND, OR, and NOT) and evaluation hierarchy

For example, to see all authors with sales in quantities greater than or equal to 75 units, or secondary authors with a royalty of greater than or equal to 60:

```
SELECT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE ta.title_id IN (SELECT title_id FROM sales
                     WHERE qty >= 75)
   OR (a.au_id IN (SELECT au_id FROM titleauthor
                 WHERE royaltyper >= 60)
      AND a.au_id IN (SELECT au_id FROM titleauthor
                    WHERE au_ord = 2));
```

See [Programming Tips and Gotchas](#) next, for more information.

Comparison operators (such as =, <, >, <=, and >=).

For example, to see author's last and first name from the **authors** table for authors who don't have a contract (i.e., contract value of zero):

```
SELECT au_lname, au_fname
FROM authors
WHERE contract = 0;
```

Lists (IN and NOT IN).

For example, to see all authors who do not yet have a title in the **titleauthors** table:

```
SELECT au_fname, au_lname
FROM authors
WHERE au_id NOT IN (SELECT au_id FROM titleauthor);
```

NULL comparisons (IS NULL and IS NOT NULL)

For example, to see titles that have NULL year-to-date sales:

```
SELECT title_id, SUBSTRING(title, 1, 25) AS title
FROM titles
WHERE ytd_sales IS NULL;
```



Be sure *not* to specify = *NULL* in a query. *NULL* is unknown and can never be equal to anything. Using = *NULL* is not the same as specifying the *IS NULL* operator. Refer to the section [IS Operator](#), earlier in this chapter, for more information.

Pattern matches (LIKE and NOT LIKE)

For example, to see authors whose last names start with a "C":

```
SELECT au_id
FROM authors
WHERE au_lname LIKE 'C%';
```

Range operations (BETWEEN and NOT BETWEEN)

For example, to see authors with last names between "Smith" and "White":

```
SELECT au_lname, au_fname
FROM authors
WHERE au_lname BETWEEN 'smith' AND 'white';
```

Programming Tips and Gotchas

The *WHERE* clause may require special handling when dealing with certain datatypes, such as *LOBs* or certain character

sets like UNICODE.

Parentheses are used to control evaluation hierarchy within a *WHERE* clause. Encapsulating a clause within parentheses tells the DBMS to evaluate that clause before others. Parentheses can be nested to create a hierarchy of evaluations. The innermost parenthetical clause will be evaluated first. You should watch parentheses very carefully for two reasons:

- You must always have an equal number of opening and closing parentheses. Any imbalance in the number of opening and closing parentheses will cause an error.
- You should be careful where you place parentheses, since misplacing a parenthesis can dramatically change the result set of your query.

For example, consider again the following query, which returns six rows in the **pubs** database on the SQL Server platform:

```
SELECT DISTINCT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE ta.title_id IN (SELECT title_id FROM sales
                     WHERE qty >= 75)
   OR (a.au_id IN (SELECT au_id FROM titleauthor
                 WHERE royaltyper >= 60)
      AND a.au_id IN (SELECT au_id FROM titleauthor
                    WHERE au_ord = 2))
```

The output from this query is as follows:

```
au_id
-----
213-46-8915
724-80-9391
899-46-2035
998-72-3567
```

Changing just one set of parenthesis produces different results:

```
SELECT DISTINCT a.au_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
WHERE (ta.title_id IN (SELECT title_id FROM sales
                     WHERE qty >= 75)
      OR a.au_id IN (SELECT au_id FROM titleauthor
                    WHERE royaltyper >= 60))
      AND a.au_id IN (SELECT au_id FROM titleauthor
                    WHERE au_ord = 2)
```

This time, the output will look like this:

au_id

213-46-8915

724-80-9391

899-46-2035

DB2

Supported as described above.

MySQL

Supported as described above.

Oracle

Supported as described above.

PostgreSQL

Supported as described above.

SQL Server

Supported as described above.

See Also

ALL/ANY/SOME
BETWEEN
DECLARE CURSOR
DELETE
EXISTS
IN
LIKE
SELECT
UPDATE



Chapter 3. SQL Statement Command Reference

This chapter is the heart of *SQL in a Nutshell*, Second Edition. It is an alphabetical listing of SQL commands with detailed explanations and examples. Each command and function is identified in [Table 3-1](#) as being "supported," "supported with variations," "supported with limitations," or "not supported," for each of the five SQL dialects covered in this book: DB2, MySQL, Oracle, PostgreSQL, SQL Server. After a brief description of the SQL2003 standard, each vendor application is discussed briefly but thoroughly, with supporting examples. If a specific platform does not support a particular command, that fact will be noted in the table that heads the command and then the command will not be listed for that platform. Similarly, although *SQL in a Nutshell*, Second Edition, is not a comprehensive review of the SQL2003 standard, each command has been validated against SQL2003.

4.1 Types of Functions

There are different ways to categorize functions into groups. The following subsections describe distinctions that are critical to understand how functions work.

4.1.1 Deterministic and Nondeterministic Functions

Functions can be either *deterministic* or *nondeterministic*. A *deterministic* function always returns the same results if given the same input values. A *nondeterministic* function may return different results every time it is called, even when the same input values are provided.

Why is it important that a given input always returns the same output? It is important because of how functions may be used within views, in user-defined functions, and in stored procedures. Restrictions vary across implementations, but these objects sometimes allow only deterministic functions within their defining code. For example, SQL Server allows the creation of an index on a column expression—as long as the expression does not contain nondeterministic functions. Rules and restrictions vary between the platforms, so check the specific documentation when using functions.

4.1.2 Aggregate and Scalar Functions

Another way of categorizing functions is in terms of whether they operate on values from just one row at a time, on values from a collection, or on a set of rows. *Aggregate functions* operate against a collection of values and return a single summarizing value. *Scalar functions* return a single value based on scalar input arguments. Some scalar functions, such as *CURRENT_TIME*, do not require any arguments.

4.1.3 Window Functions

Window functions can be thought of as being similar to aggregate functions in that they operate over many rows at one time. The difference lies in how you define those rows. Aggregate functions operate over the sets of rows defined by a query's *GROUP BY* clause. With window functions, you specify the set of rows for each function call, and different invocations of a function within the same query can execute over different sets of rows.

4.2 ANSI SQL Aggregate Functions

Aggregate functions return a single value based upon a set of other values. If used among other expressions in the item list of a *SELECT* statement, the *SELECT* must have a *GROUP BY* or *HAVING* clause. No *GROUP BY* or *HAVING* clause is required if the aggregate function is the only value retrieved by the *SELECT* statement. The supported aggregate functions and their syntax are listed in [Table 4-1](#).

Table 4-1. ANSI SQL aggregate functions

Function	Usage
AVG(<i>expression</i>)	Computes the average value of a column given by <i>expression</i> .
CORR(<i>dependent, independent</i>)	Computes a correlation coefficient.
COUNT(<i>expression</i>)	Counts the rows defined by the <i>expression</i> .
COUNT(*)	Counts all rows in the specified table or view.
COVAR_POP (<i>dependent, independent</i>)	Computes population covariance.
COVAR_SAMP(<i>dependent, independent</i>)	Computes sample covariance.
CUME_DIST(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Computes the relative rank of a hypothetical row within a group of rows, where the rank is equal to the number of rows less than or equal to the hypothetical row divided by the number of rows in the group.
DENSE_RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a dense rank (no ranks are skipped) for a hypothetical row (<i>value_list</i>) in a group of rows generated by GROUP BY.

MIN(<i>expression</i>)	Finds the minimum value in a column given by <i>expression</i> .
MAX(<i>expression</i>)	Finds the maximum value in a column given by <i>expression</i> .
PERCENT_RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.
PERCENTILE_CONT (<i>percentile</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates an interpolated value that, if added to the group, would correspond to the <i>percentile</i> given.
PERCENTILE_DISC (<i>percentile</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Returns the value with the smallest cumulative distribution value greater than or equal to <i>percentile</i> .
RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a rank for a hypothetical row (<i>value_list</i>) in a group of rows generated by GROUP BY.
REGR_AVGX(<i>dependent, independent</i>)	Computes the average of the independent variable.
REGR_AVGY(<i>dependent, independent</i>)	Computes the average of the dependent variable.
REGR_COUNT(<i>dependent, independent</i>)	Counts the number of pairs remaining in the group after any pair with one or more NULL values has been eliminated.
REGR_INTERCEPT(<i>dependent, independent</i>)	Computes the y-intercept of the least-squares-fit linear equation.

REGR_R2(<i>dependent, independent</i>)	Squares the correlation coefficient.
REGR_SLOPE(<i>dependent, independent</i>)	Determines the slope of the least-squares-fit linear equation.
REGR_SXX(<i>dependent, independent</i>)	Sums the squares of the independent variables.
REGR_SXY(<i>dependent, independent</i>)	Sums the products of each pair of variables.
REGR_SYY(<i>dependent, independent</i>)	Sums the squares of the dependent variables.
STDDEV_POP(<i>expression</i>)	Computes the population standard deviation of all <i>expression</i> values in a group.
STDDEV_SAMP(<i>expression</i>)	Computes the sample standard deviation of all <i>expression</i> values in a group.
SUM(<i>expression</i>)	Computes the sum of column values given by <i>expression</i> .
VAR_POP(<i>expression</i>)	Computes the population variance of all <i>expression</i> values in a group.
VAR_SAMP(<i>expression</i>)	Computes the sample standard deviation of all <i>expression</i> values in a

expression)

group.

Technically speaking, *ANY*, *EVERY*, and *SOME* are considered aggregate functions. However, they have been discussed as range search criteria since they are most often used that way. Refer to [ALL/ANY/SOME Operators](#) for more information on these functions.

The number of values processed by an aggregate function varies depending on the number of rows queried from the table. This behavior makes aggregate functions different from scalar functions, which can only operate on the values of a single row per invocation.

The general syntax of an aggregate function is:

aggregate_function_name ([*ALL* | *DISTINCT*] *expression*)

The aggregate function name may be *AVG*, *COUNT*, *MAX*, *MIN*, or *SUM*, as listed in [Table 4-1](#). The *ALL* keyword, which specifies the default behavior, evaluates all rows when aggregating the value of the function. The *DISTINCT* keyword uses only distinct values when evaluating the function.



All aggregate functions except *COUNT(*)* will ignore NULL values when computing their results.

AVG and SUM

The *AVG* function computes the average of values in a column or an expression. *SUM* computes the sum. Both functions work with numeric values and ignore NULL values. Use the *DISTINCT* keyword to compute the average or sum of all distinct values of a column or expression.

SQL Standard Syntax

AVG ([*ALL* | *DISTINCT*] *expression*)

SUM ([*ALL* | *DISTINCT*] *expression*)

MySQL, PostgreSQL, and SQL Server

All these platforms support the SQL2003 syntax of *AVG* and *SUM*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and the following analytic syntax:

AVG ([*ALL* | *DISTINCT*] *expression*) *OVER* (*window_clause*)

SUM ([*ALL* | *DISTINCT*] *expression*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the [Section 4.3](#) later in this chapter.

Examples

The following query computes average year-to-date sales for each type of book:

```
SELECT type, AVG( ytd_sales ) AS "average_ytd_sales"FROM titles GROUP BY type;
```

This query returns the sum of year-to-date sales for each type of book:

```
SELECT type, SUM( ytd_sales ) FROM titles GROUP BY type;
```

CORR

The *CORR* function returns the correlation coefficient between a set of dependent and independent variables.

SQL2003 Syntax

Calls the function with two variables, one dependent and the other independent:

```
CORR( dependent, independent )
```

Any pair in which either the dependent variable, independent variable, or both are NULL is ignored. The result of the function is NULL when none of the input pairs consist of two non-NULL values.

Oracle

Oracle supports the SQL2003 syntax, and the following analytic syntax:

```
CORR ( dependent, independent ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the [Section 4.3](#) later in this chapter.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *CORR* function.

Example

The following *CORR* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;
```

Y	X
1	3
2	2
3	1

```
SELECT CORR(y,x) FROM test2;
```

```
CORR(Y,X)
-----
-1
```

COUNT

The *COUNT* function is used to compute the number of rows in an expression.

SQL2003 Syntax

COUNT(*)

COUNT([ALL|DISTINCT] *expression*)

COUNT(*)

Counts all the rows in the target table whether or not they include NULLs.

COUNT([ALL|DISTINCT] *expression*)

Computes the number of rows with non-NULL values in a specific column or expression. When the keyword *DISTINCT* is used, duplicate values will be ignored and a count of the distinct values is returned. *ALL* returns the number of non-NULL values in the expression and is implicit when *DISTINCT* is not used.

MySQL, PostgreSQL, and SQL Server

All of these platforms support the SQL2003 syntax of *COUNT*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and the following analytic syntax:

COUNT ({*|[DISTINCT] *expression*}) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

Examples

This query counts all rows in a table:

```
SELECT COUNT(*) FROM publishers;
```

The following query finds the number of different countries where publishers are located:

```
SELECT COUNT(DISTINCT country) "Count of Countries"  
FROM publishers
```

COVAR_POP

The *COVAR_POP* function returns the population covariance of a set of dependent and independent variables.

SQL2003 Syntax

Call the function with two variables, one dependent and the other independent:

COVAR_POP(*dependent*, *independent*)

The function disregards any pair in which either the dependent variable, independent variable, or both are NULL. If no rows remain in the group after NULL elimination, then the result of the function is NULL.

Oracle

Oracle supports the SQL2003 syntax and implements the following analytic syntax:

COVAR_POP (*dependent*, *independent*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

DB2

In DB2, the function is named *CORRELATION*.

MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *COVAR_POP* function.

Example

The following *COVAR_POP* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;
```

```
      Y      X
-----
      1      3
      2      2
      3      1
```

```
SELECT COVAR_POP(y,x) FROM test2;
```

```
COVAR_POP(Y,X)
```

```
-----
-.66666667
```

COVAR_SAMP

The *COVAR_SAMP* function returns the sample covariance of a set of dependent and independent variables.

SQL2003 Syntax

Call the function with two variables, one dependent and the other independent:

COVAR_SAMP(*dependent*, *independent*)

The function disregards any pair in which either the dependent variable, independent variable, or both are NULL. The result of the function is NULL when none of the input pairs consist of two non-NULL values.

Oracle

Oracle supports the SQL2003 syntax and implements the following analytic syntax:

COVAR_SAMP (*dependent*, *independent*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *COVAR_SAMP* function.

Example

The following *COVAR_SAMP* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;
```

```
   Y   X
-----
   1   3
   2   2
   3   1
```

```
SQL> SELECT COVAR_SAMP(y,x) FROM test2;
```

```
COVAR_SAMP(Y,X)
-----
             -1
```

CUME_DIST

Computes the relative rank of a hypothetical row within a group of rows, where that relative rank is computed as follows:

$$(rows_preceding_hypothetical + rows_peered_with_hypothetical) / rows_in_group$$

Bear in mind that the *rows_in_group* value includes the hypothetical row that you are proposing when you call the function.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [,expression...]
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
CUME_DIST OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *CUME_DIST* aggregate function.

Example

The following example determines the relative rank of the hypothetical new row (*num=4, odd=1*) within each group of rows from **test4**, where groups are distinguished by the values in the *odd* column:

SELECT * FROM test4;

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

SELECT odd, CUME_DIST(4,1) WITHIN GROUP (ORDER BY num, odd)

FROM test4

GROUP BY odd;

ODD	CUME_DIST(4,1) WITHIN GROUP (ORDER BY NUM, ODD)
0	1
1	.8

In group *odd=0*, the new row comes after the three rows: (0,0), (2,0), and (4,0). It will peer with itself. The total number of rows in the group will be four, which includes the hypothetical row. The relative rank, therefore, is computed as follows:

$$\begin{aligned} & (3 \text{ rows preceding} + 1 \text{ peering}) / (3 \text{ in group} + 1 \text{ hypothetical}) \\ & = 4 / 4 = 1 \end{aligned}$$

In group *odd=1*, the new row follows the three rows (1,1), (3,1), and a duplicate (3,1). Again, there is one peer, the hypothetical row itself. The number of rows in the group is five, which includes the hypothetical row. The relative rank is then:

$$\begin{aligned} & (3 \text{ rows preceding} + 1 \text{ peering}) / (4 \text{ in group} + 1 \text{ hypothetical}) \\ & = 4 / 5 = .8 \end{aligned}$$

DENSE_RANK

Computes a rank in a group for a hypothetical row that you supply. This is a *dense rank*. Rankings are never skipped, even when a group contains rows that rank identically.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [,expression...]
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
DENSE_RANK( ) OVER (partitioning ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *DENSE_RANK* aggregate function. However, DB2 does support *DENSE_RANK* as an analytic function. See the section later in this chapter titled [Section 4.3](#).

Example

The following example determines the dense rank of the hypothetical new row (*num=4, odd=1*) within each group of rows from **test4**, where groups are distinguished by the values in the odd column:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, DENSE_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
```

```
FROM test4
```

```
GROUP BY odd;
```

```
ODD DENSE_RANK(4,1) WITHIN GROUP (ORDER BY NUM, ODD)
```

0	4
1	3

In group `odd=0`, the new row comes after (0,0), (2,0), and (4,0), and thus it is position 4. In group `odd=1`, the new row follows (1,1), (3,1), and a duplicate (3,1). In that case, the duplicate occurrences of (3,1) both rank #2, so the new row is ranked #3. Compare this behavior with *RANK*, which gives a different result.

MIN and MAX

MIN(expression) and *MAX(expression)* find the minimum and maximum value of *expression* (string, datetime, or numeric) in a set of rows. *DISTINCT* or *ALL* may be used with these functions, but do not affect the result.

SQL2003 Syntax

MIN([ALL | DISTINCT] *expression*)

MAX([ALL | DISTINCT] *expression*)

PostgreSQL and SQL Server

These platforms support the SQL2003 syntax of *MIN* and *MAX*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and implements the following analytic syntax:

MIN ({ALL|[DISTINCT] *expression*}) *OVER* (*window_clause*)

MAX ({ALL|[DISTINCT] *expression*}) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

MySQL

MySQL supports the SQL2003 syntax of *MIN* and *MAX*. MySQL also supports the functions *LEAST*() and *GREATEST*(), providing the same capabilities.

Examples

The following query finds the best and worst sales for any title on record:

```
SELECT MIN(ytd_sales), MAX(ytd_sales)
FROM titles;
```

Aggregate functions are used often in the *HAVING* clause of queries with *GROUP BY*. The following query selects all categories (types) of books that have an average price for all books in the category higher than \$15.00:

```
SELECT type 'Category', AVG( price ) 'Average Price'
FROM titles
GROUP BY type
HAVING AVG(price) > 15
```

PERCENT_RANK

Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [,expression...]
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following syntax:

```
PERCENT_RANK( ) OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *PERCENT_RANK* aggregate function.

Example

The following example determines the percentage rank of the hypothetical new row (*num=4, odd=1*) within each group of rows from **test4**, where groups are distinguished by the values in the odd column:

```
SELECT * FROM test4;
```

```
      NUM      ODD
-----
      0         0
      1         1
      2         0
      3         1
      3         1
      4         0
      5         1
```

```
SELECT odd, PERCENT_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
```

```
FROM test4
```

```
GROUP BY odd;
```

```
      ODD PERCENT_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
```

```
-----
      0                1
      1                .75
```

In group `odd=0`, the new row comes after (0,0), (2,0), and (4,0), and thus it is position 4. The rank computation is: $(4^{\text{th}} \text{rank} - 1) / 3 \text{ rows} = 100\%$. In group `odd=1`, the new row follows (1,1), (3,1), and a duplicate (3,1), and is again ranked at #4. The rank computation for `odd=1` is: $(4^{\text{th}} \text{rank} - 1) / 4 \text{ rows} = 3/4 = 75\%$.

PERCENTILE_CONT

Generates an interpolated value corresponding to a percentile that you specify.

SQL2003 Syntax

In the following syntax, *percentile* is a number between zero and one:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_list)
```

```
sort_list ::= sort_item [,sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_CONT (percentile) WITHIN GROUP
```

```
(ORDER BY sort_list) OVER (partitioning)
```

See [Section 4.3](#) later in this chapter for a description of partitioning.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_CONT*.

Example

The following example groups the data in `test4` by the column named `odd`, and invokes *PERCENTILE_CONT* to return a 50th percentile value for each group:

```
SELECT * FROM test4;
```

```
NUM  ODD
-----
0    0
1    1
2    0
3    1
3    1
4    0
5    1
```

```
SELECT odd, PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
FROM test4
GROUP BY odd;
      ODD PERCENTILE_CONT(0.50)WITHINGROUP(ORDERBYNUM)
```

```
-----
      0                2
      1                3
```

PERCENTILE_DISC

Determines the value in a group with the smallest cumulative distribution greater than or equal to a percentile that you specify.

SQL2003 Syntax

In the following syntax, *percentile* is a number between zero and one:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_list)
sort_list ::= sort_item [,sort_item...]
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_DISC (percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See [Section 4.3](#) later in this chapter for a description of partitioning.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_DISC*.

Example

The following example is similar to that for *PERCENTILE_CONT*, except that it returns, for each group, the value closest, but not exceeding, the 60th percentile:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, PERCENTILE_DISC(0.60) WITHIN GROUP (ORDER BY NUM)
```

```
FROM test4
```

```
GROUP BY odd;
```

```
PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
```

2
3

RANK

Computes a rank in a group for a hypothetical row that you supply. This is not a dense rank. If the group contains rows that rank identically, then it's possible for ranks to be skipped. If you want a dense rank, use the *DENSE_RANK* function.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
```

```
value_list ::= expression [, expression...]
```

```
sort_list ::= sort_item [, sort_item...]
```

```
sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
RANK( ) OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled [Section 4.3](#).

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *RANK* aggregate function.

Example

The following example determines the rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from `test4`, where groups are distinguished by the values in the `odd` column:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
```

```
FROM test4
```

```
GROUP BY odd;
```

ODD	RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
0	4
1	4

In both cases, the rank of the hypothetical new row is 4. In group `odd=0`, the new row comes after: `(0,0)`, `(2,0)`, and `(4,0)`, and thus it is position 4. In group `odd=1`, the new row follows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`. In that case, the new row is preceding by three rows, so it is ranked #4. Compare this behavior with `DENSE_RANK`.

The REGR Family of Functions

SQL2003 defines a family of functions, having names beginning with `REGR_`, that relate to different aspects of linear regression. The functions work in the context of a least-squares regression line.

SQL2003 Syntax

Following is the syntax and a brief description of each `REGR_` function:

```
REGR_AVGX( dependent, independent )
```

Averages (as in `AVG(x)`) the *independent* variable values.

```
REGR_AVGY( dependent, independent )
```

Averages (as in `AVG(y)`) the *dependent* variable values.

```
REGR_COUNT( dependent, independent )
```

Counts the number of non-NULL number pairs.

`REGR_INTERCEPT(dependent, independent)`

Computes the y-intercept of the regression line.

`REGR_R2(dependent, independent)`

Computes the coefficient of determination.

`REGR_SLOPE(dependent, independent)`

Computes the slope of the regression line.

`REGR_SXX(dependent, independent)`

Sums the squares of the independent variable values.

`REGR_SXY(dependent, independent)`

Sums the products of each pair of values.

`REGR_SYY(dependent, independent)`

Sums the squares of the dependent variable values.

The `REGR_` functions only work on number pairs containing two non-NULL values. Any number pair with one or more NULL values will be ignored.

DB2 and Oracle

DB2 and Oracle support the SQL2003 syntax for all `REGR_` functions. In addition, DB2 allows the shortened name `REGR_ICPT` in place of `REGR_INTERCEPT`.

Oracle supports the following analytic syntax:

`REGR_function (dependent, independent) OVER (window_clause)`

For an explanation of the `window_clause`, see the section later in this chapter titled [Section 4.3](#).

MySQL, PostgreSQL, and SQL Server

These platforms do not implement the `REGR` family of functions.

Example

The following `REGEXP_COUNT` example demonstrates that any pair with one or more NULL values is ignored. The table `test3` contains three non-NULL number pairs, and three other pairs having at least one NULL:

SQL> SELECT * FROM test3;

Y	X
1	3
2	2
3	1
4	NULL
NULL	4
NULL	NULL

The *REGR_COUNT* function ignores the pairs having NULLs, counting only those pairs with non-NULL values:

```
SELECT REGR_COUNT(y,x) FROM test3;
```

```
REGR_COUNT(Y,X)
```

```
-----
```

```
3
```

Likewise, all other *REGR_* functions filter out any pairs having NULL values before performing their respective computations.

STDDEV_POP

Use *STDDEV_POP* to find the *population standard deviation* within a group of numeric values.

SQL2003 Syntax

```
STDDEV_POP( numeric_expression )
```

DB2 and MySQL

Use the *STDDEV* function. In DB2 and MySQL, *STDDEV* returns the population standard deviation.

PostgreSQL

This platform does not provide a function to compute population standard deviation.

Oracle

Oracle supports the standard syntax and the following analytic syntax:

```
STDDEV_POP( numeric_expression ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

SQL Server

Use the *STDEV* function.

Example

The following example computes the population standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;
```

```
X
-----
1
2
3

SELECT STDDEV_POP(x) FROM test;

STDDEV_POP(X)
-----
.816496581
```

STDDEV_SAMP

Use *STDDEV_SAMP* to find the *sample standard deviation* within a group of numeric values.

SQL2003 Syntax

STDDEV_SAMP(*numeric_expression*)

Oracle

Oracle supports the standard syntax. Oracle also provides the *STDDEV* function, which operates similar to *STDDEV_SAMP* except that it returns zero as not NULL when there is only one value in the set.

Oracle also supports analytic syntax:

STDDEV_SAMP (*numeric_expression*) OVER (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

DB2

This platform does not provide a function to compute sample standard deviation.

MySQL

MySQL does not provide a function to compute sample standard deviation. MySQL does provide a function named *STDDEV*, but it returns the *population* standard deviation.

PostgreSQL

Use *STDDEV*.

SQL Server

Use *STDEV* (with only one *D!*).

Example

The following example computes the sample standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;
```

```
X
-----
1
2
3

SELECT STDDEV_SAMP(x) FROM test;
STDDEV_SAMP(X)
-----
1
```

VAR_POP

Use *VAR_POP* to compute the population variance of a set of values.

SQL2003 Syntax

VAR_POP(*numeric_expression*)

DB2 and PostgreSQL

These platforms do not provide a function to compute population variance.

MySQL

Use the *VARIANCE* function, which in MySQL returns the population variance.

Oracle

Oracle supports the standard syntax and the following analytic syntax:

VAR_POP (*numeric_expression*) *OVER* (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

SQL Server

Use the *VARP* function.

Example

The following example computes the population variance for the values 1, 2, and 3:

```
SELECT * FROM test;
```

```
X
-----
1
2
3
```

```
SELECT VAR_POP(x) FROM test;
```

```
VAR_POP(X)
-----
.666666667
```

VAR_SAMP

Use *VAR_SAMP* to compute the sample variance of a set of values.

SQL2003 Syntax

```
VAR_SAMP( numeric_expression )
```

DB2 and PostgreSQL

Use *VARIANCE(numeric_expression)* to compute sample variance.

MySQL

MySQL provides no function for computing sample variance. There is the *VARIANCE* function, but in MySQL that function returns the *population* variance.

Oracle

Oracle supports the standard syntax. You may also use the *VARIANCE* function, which differs from *VAR_SAMP* by returning zero (and not NULL) for sets having only a single value.

Oracle also supports analytic syntax:

```
VAR_SAMP( numeric_expression ) OVER ( window_clause )
```

For an explanation of the *window_clause*, see the section later in this chapter titled [Section 4.3](#).

SQL Server

Use the *VAR* function.

Example

The following example computes the sample variance for the values 1, 2, and 3:

```
SELECT * FROM test;
```

X

1

2

3

SELECT VAR_SAMP(x) FROM test;

VAR_SAMP(X)

1



< Day Day Up >



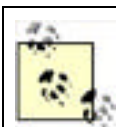
4.3 ANSI SQL Window Functions

SQL2003 allows for a *window_clause* in aggregate function calls, the addition of which makes those functions into window functions. Both Oracle and DB2 support this window function syntax. This section describes how to use the *window_clause* within Oracle and DB2.



Oracle tends to refer to window functions as *analytic functions*.

Window, or analytic, functions are similar to standard aggregate functions in that they operate on multiple rows, or groups of rows, within the result set returned from a query. However, the groups of rows that a window function operates on are defined not by a GROUP BY clause, but by partitioning and windowing clauses. Furthermore, the order within these groups is defined by an ordering clause, but that order only affects function evaluation, and has no effect on the order in which rows are returned by the query.



Window functions are the last items in a query to be evaluated except for the *ORDER BY* clause. Because of this late evaluation, window functions *cannot* be used within the *WHERE*, *GROUP BY*, or *HAVING* clauses.

4.3.1 SQL2003's Window Syntax

SQL2003 specifies the following syntax for window functions:

FUNCTION_NAME(*expr*) OVER {*window_name*}(*window_specification*)}

window_specification ::= [*window_name*][*partitioning*][*ordering*][*framing*]

partitioning ::= PARTITION BY *value* [, *value*...] [COLLATE *collation_name*]

ordering ::= ORDER [SIBLINGS] BY *rule* [, *rule*...]

rule ::= {*value*|*position*|*alias*} [ASC|DESC] [NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {*start*|*between*} [*exclusion*]

start ::= {UNBOUNDED PRECEDING|*unsigned-integer* PRECEDING|CURRENT ROW}

between ::= BETWEEN *bound* AND *bound*

bound ::= {*start*|UNBOUNDED FOLLOWING|*unsigned-integer* FOLLOWING}

exclusion ::= {EXCLUDE CURRENT ROW|EXCLUDE GROUP
|EXCLUDE TIES|EXCLUDE NO OTHERS}

4.3.2 Oracle's Window Syntax

Oracle's window function syntax is as follows:

FUNCTION_NAME(*expr*) OVER (*window_clause*)

window_clause ::= [*partitioning*] [*ordering* [*framing*]]

partitioning ::= PARTITION BY *value* [, *value*...]

ordering ::= ORDER [SIBLINGS] BY *rule* [, *rule*...]

rule ::= {*value*|*position*|*alias*} [ASC|DESC]
[NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {*not_range*|*begin* AND *end*}

not_range ::= {UNBOUNDED PRECEDING
|CURRENT ROW
|*value* PRECEDING}

begin ::= {UNBOUNDED PRECEDING
|CURRENT ROW
|*value* {PRECEDING|FOLLOWING}}

end ::= {UNBOUNDED FOLLOWING
|CURRENT ROW
|*value* {PRECEDING|FOLLOWING}}

4.3.3 DB2's Window Syntax

DB2's syntax is similar to Oracle's. For OLAP, ranking, and numbering functions, DB2 allows the following syntax:

FUNCTION_NAME(*expr*) OVER (*window_clause*)

window_clause ::= [*partitioning*] [*ordering*]

partitioning ::= PARTITION BY (*value* [, *value*...])

ordering ::= {ORDER BY *rule* [, *rule*...] | ORDER OF *table_name*}

rule ::= {*value*|*position*|*alias*} [ASC|DESC [NULLS {FIRST|LAST}]]

When aggregate functions (e.g. AVG) are used as window functions, DB2 allows the addition of a *framing* clause:

FUNCTION_NAME(*expr*) OVER (*window_clause*)

window_clause ::= [*partitioning*] [*ordering* [*framing*]] [*all*/*framing*]

all ::= RANGE UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

partitioning ::= PARTITION BY (*value* [, *value*...])

ordering ::= {ORDER BY *rule* [, *rule*...] | ORDER OF *table_name*}

rule ::= {*value*|*position*|*alias*} [ASC|DESC [NULLS {FIRST|LAST}]]

framing ::= {ROWS|RANGE} {*group_start*|*group_between*|*group_end*}

group_start ::= {UNBOUNDED PRECEDING|*unsigned-integer* PRECEDING
|CURRENT ROW}

group_between ::= BETWEEN {UNBOUNDED PRECEDING|*unsigned_integer* PRECEDING
|*unsigned_integer* FOLLOWING|CURRENT ROW}
AND {UNBOUNDED FOLLOWING|*unsigned_integer* PRECEDING
|*unsigned_integer* FOLLOWING|CURRENT ROW}

group_end ::= UNBOUNDED FOLLOWING|*unsigned-integer* FOLLOWING}

4.3.4 Partitioning

Partitioning the rows operated on by the partition clause is similar to using the *GROUP BY* expression on a standard *SELECT* statement. The partitioning clause takes a list of expressions that will be used to divide the result set into groups. We'll use the following table as the basis for some examples:

SELECT * FROM odd_nums;

NUM	ODD
0	0
1	1
2	0
3	1

The following results illustrate the effects of partitioning by **ODD**. The sum of the even numbers is 2 (0+2), and the sum of the odd numbers is 4 (1+3). The second column of the result set reports the sum of all values in the partition to which *that row* belongs. Yet all the detail rows are returned. The query provides summary results in the context of detail rows:

```
SELECT NUM, SUM(NUM) OVER (PARTITION BY ODD) S  
FROM ODD_NUMS;
```

NUM	S
0	2
2	2
1	4
3	4

Not using a partitioning clause at all will sum all of the numbers in the **NUM** column for each row returned by the query. In effect, the entire result set is treated as a single, large partition:

```
SELECT NUM, SUM(NUM) OVER ( ) S FROM ODD_NUMS;
```

NUM	S
0	6
1	6
2	6
3	6

4.3.5 Ordering

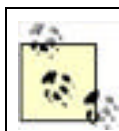
You specify the order of the rows on which an analytic function operates using the *ordering* clause. However, this analytic ordering clause does not define the result set ordering. To define the overall result set ordering, you must use the query's *ORDER BY* clause. The following use of Oracle's *FIRST_VALUE* function illustrates the effect of different orderings of the partitions:

```
SELECT NUM,  
      SUM(NUM) OVER (PARTITION BY ODD) S,  
      FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM ASC) first_asc,
```

```
FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM DESC) first_desc  
FROM ODD_NUMS;
```

NUM	S	FIRST_ASC	FIRST_DESC
0	2	0	2
2	2	0	2
1	4	1	3
3	4	1	3

As you can see, the *ORDER BY* clauses in the window function invocations affect the ordering of the rows in the respective partitions when those functions are evaluated. Thus, *ORDER BY NUM ASC* orders partitions in ascending order, resulting in **0** for the first value in the even-number partition and **1** for the first value in the odd-number partition. *ORDER BY NUM DESC* has the opposite effect.



The preceding query also illustrates an important point: using window functions, you can summarize and order many different ways in the same query.

4.3.6 Grouping or Windowing

Many analytic functions also allow you to specify a virtual, moving window surrounding a row within a partition. You do this using the *framing* clause. Such moving windows are useful for running calculations such as a running total.

The following, Oracle-based example uses the *framing* clause on the analytic variant of *SUM* to calculate a running sum of the values in the first column. No partitioning clause is used, so each invocation of *SUM* operates over the entire result set. However, the *ORDER BY* clause sorts the rows for *SUM* in ascending order of **NUM**'s value, and the *BETWEEN* clause (which is the windowing clause) causes each invocation of *SUM* to include values for **NUM** only up through the current row. Each successive invocation of *SUM* includes yet another value for **NUM**, in order, from the lowest value of **NUM** to the greatest:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS  
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS;
```

NUM	S
0	0
1	1
2	3
3	6

This example's a bit too easy, as the order of the final result set happens to match the order of the running total. That doesn't need to be the case. The following example generates the same results, but in a different order. You can see that the running total values are appropriate for each value of **NUM**, but the rows are presented in a different order than before. The result set ordering is completely independent of the ordering used for window function calculations:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS  
BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS  
ORDER BY NUM DESC;
```

NUM	S
3	6
2	3
1	1
0	0

4.3.7 List of Window Functions

SQL2003 specifies that any aggregate function may also be used as a window function. Both Oracle and DB2 largely follow the standard in that respect, so you'll find that you can take just about any aggregate function (certainly the standard ones) and apply to it the window function syntax described in the preceding sections.

In addition to the aggregate functions, SQL2003 defines the window functions described in the following sections. Only Oracle and DB2 currently implement these functions. All examples use the following table and data, which is a variation on the **ODD_NUMS** table used earlier to illustrate the concepts of partitioning, ordering, and grouping:

SELECT * FROM test4;

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

Platform-specific window functions for Oracle (there are none for DB2) are included in the lists found under [Section 4.5](#) later in this chapter.

CUME_DIST()

Calculates the cumulative distribution, or relative rank, of the current row to other rows in the same partition. The calculation for a given row is as follows:

$\text{number of peer or preceding rows} / \text{number of rows in partition}$

Because the result for a given row depends on the number of rows preceding that row in the same partition, it's important to always specify an *ORDER BY* clause when invoking this function.

SQL2003 Syntax

CUME_DIST() OVER {*window_name*}{*window_specification*}

DB2

DB2 does not support the *CUME_DIST()* window function.

Oracle

Oracle does not allow the framing portion of the windowing syntax. Oracle requires the *ordering* clause:

CUME_DIST() OVER (*[partitioning]* *ordering*)

Example

The following Oracle-based example uses *CUME_DIST()* to generate a relative rank for each row, ordering by **NUM**, after partitioning the data by **ODD**:

```
SELECT NUM, ODD, CUME_DIST( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
```

```
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	.333333333
2	0	.666666667
4	0	1
1	1	.25
3	1	.75
3	1	.75
5	1	1

Following is an explanation of the calculation behind the rank for the row in which **NUM=0**:

1. Because of the *ORDER BY* clause, the rows in the partition are ordered as follows:

```
NUM=0
NUM=2
NUM=4
```

1. There are no rows preceding **NUM=0**.
2. There is one row that is a peer of **NUM=0**, and that is the **NUM=0** row itself. Thus, the divisor is 1.
3. There are three rows in the partition as a whole, making the dividend 3.
4. The result of 1/3 is .33 repeating, as shown in the example output.

DENSE_RANK()

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question, and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same. Unlike the case with *RANK()*, gaps in rank numbers will not result from two rows sharing the same rank.

SQL2003 Syntax

```
DENSE_RANK( ) OVER {window_name}(window_specification)}
```

DB2

DB2 requires the *ordering* clause and does not allow the *framing* clause:

```
DENSE_RANK( ) OVER ([partitioning] ordering)
```

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

```
DENSE_RANK( ) OVER ([partitioning] ordering)
```

Example

Compare the results from the following Oracle-based example to those shown in the section on the *RANK()* function:

```
SELECT NUM, DENSE_RANK( ) OVER (ORDER BY NUM) rank
```

```
FROM test4;
```

NUM	RANK
0	1
1	2
2	3
3	4
3	4
4	5
5	6

The two rows where NUM=3 are both ranked at #3. The next higher row is ranked at #4. Rank numbers are not skipped, hence the term "dense."

RANK()

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question, and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same, and will lead to subsequent gaps in rank numbers.

SQL2003 Syntax

```
RANK( ) OVER {window_name}(window_specification)}
```

DB2

DB2 requires the *ordering* clause and does not allow the *framing* clause:

`RANK() OVER ([partitioning] ordering)`

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

`RANK() OVER ([partitioning] ordering)`

Example

The following Oracle-based example uses the **NUM** column to rank the rows in the **test4** table:

SELECT NUM, RANK() OVER (ORDER BY NUM) rank

FROM test4;

NUM	RANK
0	1
1	2
2	3
3	4
3	4
4	6
5	7

Because both rows where **NUM**=3 rank the same at #4, the next higher row will be ranked at #6. The #5 rank is skipped.

PERCENT_RANK

Computes the relative rank of a row by dividing that row's rank less 1 by the number of rows in the partition, also less 1:

$(rank - 1) / (rows - 1)$

Compare this calculation to that used for *CUME_DIST*.

SQL2003 Syntax

`PERCENT_RANK() OVER ({window_name}(window_specification))`

DB2

DB2 does not support the *PERCENT_RANK()* window function.

Oracle

Oracle requires the *ordering* clause and does not allow the *framing* clause:

`PERCENT_RANK() OVER ([partitioning] ordering)`

Example

The following, Oracle-based example assigns a relative rank to values of **NUM**, partitioning the data on the **ODD** column:

```
SELECT NUM, ODD, PERCENT_RANK( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	0
2	0	.5
4	0	1
1	1	0
3	1	.333333333
3	1	.333333333
5	1	1

Following is an explanation of the calculation behind the rank for the row in which **NUM**=2:

1. Row **NUM**=2 is the second row in its partition; thus, it ranks #2.
2. Subtract 1 from 2 to get a divisor of 1.
3. The dividend is the total number of rows in the partition, or 3.
4. Subtract 1 from 3 to get a dividend of 2.
5. The result of 1/3 is .33 repeating, as shown in the example.

ROW_NUMBER

Assigns a unique number to each row in a partition.

SQL2003 Syntax

`ROW_NUMBER() OVER ({window_name}(window_specification))`

DB2

DB2 does not allow the *framing* clause, and it makes the *ordering* clause optional:

`ROW_NUMBER() OVER ([partitioning] [ordering])`

Oracle

Oracle requires the *ordering* clause and does not allow the *framing* clause:

`ROW_NUMBER() OVER ([partitioning] ordering)`

Example

```
SELECT NUM, ODD, ROW_NUMBER( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	1
2	0	2
4	0	3
1	1	1
3	1	2
3	1	3
5	1	4



4.4 ANSI SQL Scalar Functions

The ANSI SQL scalar functions return a single value each time they are invoked. The SQL standard provides many scalar functions that can be used to manipulate date and time types, strings, and numbers, as well as retrieve system information such as the current user or login name. Scalar functions fall into the categories listed in [Table 4-2](#).

Table 4-2. Categories of scalar functions

Function category	Explanation
Built-in	Performs operations on values or settings built into the database. Oracle uses the term "built-in" to describe all the specialty functions that are provided by Oracle, and thus "built into" their DBMS. This is a distinct and separate usage from the "built-in" functions described here.
CASE and CAST	While these two functions operate on scalar input values, they are in a category all their own. <i>CASE</i> supplies <i>IF-THEN</i> logic to SQL statements and <i>CAST</i> can convert values from one datatype to another.
Date and Time	Performs operations on <i>temporal</i> datatypes and returns values in a <i>temporal</i> datatype format. There is no SQL2003 function that operates on a temporal datatype and returns a temporal result. The closest function is <i>EXTRACT</i> (covered in Section 4.4.3 later in this chapter), which operates on temporal values and returns numeric values. Functions returning temporal values but operating on no arguments are covered in the later Section 4.4.1 .
Numeric	Performs operations on numeric values and returns numeric values.
String	Performs operations on character values (e.g., <i>CHAR</i> , <i>VARCHAR</i> , <i>NCHAR</i> , <i>NVARCHAR</i> , and <i>CLOB</i>) and returns a string or numeric value.

4.4.1 Built-in Scalar Functions

SQL2003 built-in scalar functions identify both the current user session and the characteristics of the current user session, such as the current session privileges. Built-in scalar functions are always nondeterministic. The *CURRENT_DATE*, *CURRENT_TIME*, and *CURRENT_TIMESTAMP* functions listed in [Table 4-3](#) are built-in functions that fall into the date-and-time category of functions. Although the five platforms provide many additional functions beyond these SQL built-ins, the SQL standard defines only those listed in [Table 4-3](#).

Table 4-3. ANSI SQL built-in scalar functions

Function	Usage
<i>CURRENT_DATE</i>	Returns the current date.
<i>CURRENT_TIME</i>	Returns the current time.
<i>CURRENT_TIMESTAMP</i>	Returns the current date and time.
<i>CURRENT_USER</i> or <i>USER</i>	Returns the currently active user within the database server.
<i>SESSION_USER</i>	Returns the currently active Authorization ID, if it differs from the user.
<i>SYSTEM_USER</i>	Returns the currently active user within the host operating system.

DB2

DB2 supports the SQL2003 built-in scalar functions *CURRENT_DATE*, *CURRENT_TIME*, *CURRENT_TIMESTAMP*, *CURRENT_USER*, and *USER*. The SQL2003 scalar functions *CURRENT_ROLE*, *LOCAL_TIME*, *LOCAL_TIMESTAMP*, *SESSION_USER*, and *SYSTEM_USER* are not supported.

MySQL

MySQL supports all the SQL2003 built-in scalar functions, plus Oracle's variant *SYSDATE*. In addition, MySQL supports *NOW()* as a synonym of the function *CURRENT_TIMESTAMP*.

Oracle

Oracle supports *USER* and *SYSDATE* as a synonym of *CURRENT_TIMESTAMP*.

PostgreSQL

PostgreSQL supports all the SQL2003 built-in scalar functions except *SESSION_USER*.

SQL Server

SQL Server supports all the built-in scalar functions.

Examples

The following queries retrieve the values from built-in functions. Notice that the various platforms return dates in their native formats:

```
/* On MySQL */
```

```
SELECT CURRENT_TIMESTAMP;'2001-12-15 23:50:26'
```

```
/* DB2 */
```

```
VALUES CURRENT_TIMESTAMP'2001-12-15 23.50.26.000000'
```

```
/* On Microsoft SQL Server */
```

```
SELECT CURRENT_TIMESTAMPGO'Dec 15,2001 23:50:26'
```

```
/* On Oracle */
```

```
SELECT USER FROM dual;dylan
```

4.4.2 CASE and CAST Functions

ANSI SQL2003 provides a function named *CASE* that can be used to create *IF-THEN* flow-control logic within queries or update statements. The *CAST* function is for converting between datatypes and is also in the ANSI standard. All of the databases covered by this book provide ANSI standard support for both the *CASE* and *CAST* functions.

CASE

The *CASE* function provides *IF-THEN-ELSE* functionality within a *SELECT* or *UPDATE* statement. It evaluates a list of conditions and returns one value out of several possible values.

CASE has two usages: simple and searched. Simple *CASE* expressions compares one value, the *input_value*, with a list of other values and return a result associated with the first matching value. Searched *CASE* expressions allow the analysis of several logical conditions and return a result associated with the first one that is true.

All vendors provide the ANSI SQL2003 syntax for *CASE*.

SQL2003 Syntax and Description

```
-- Simple comparison operation
```

```
CASE input_value WHEN when_condition THEN resulting_value[...n]
```

```
[ELSE else_result_value]
```

```
END

-- Boolean searched operation

CASE

WHEN Boolean_condition THEN resulting_value[...n]

[ELSE else_result_expression]

END
```

In the simple *CASE* function, the *input_value* is evaluated against each *WHEN* clause. The *resulting_value* is returned for the first TRUE instance of *input_value = when_condition*. If no *when_condition* evaluates as TRUE, the *else_result_value* is returned. If no *else_result_value* is specified, then NULL is returned.

In the more elaborate Boolean searched operation, the structure is essentially the same as the simple comparison operation except that each *WHEN* clause has its own Boolean comparison operation.

In either usage, multiple *WHEN* clauses are used, though only one *ELSE* clause is necessary.

Examples

Here is a simple comparison operation where the *CASE* function alters the display of the **contract** column to make it more understandable:

```
SELECT au_fname,
       au_lname,
       CASE contract
         WHEN 1 THEN 'Yes'
         ELSE 'No'
       END 'contract'
FROM   authors
WHERE  state = 'CA'
```

Here is an elaborate searched *CASE* function in a *SELECT* statement that will report how many titles have been sold in different year-to-date sales ranges:

```
SELECT CASE
         WHEN ytd_sales IS NULL THEN 'Unknown'
         WHEN ytd_sales <= 200 THEN 'Not more than 200'
         WHEN ytd_sales <= 1000 THEN 'Between 201 and 1000'
         WHEN ytd_sales <= 5000 THEN 'Between 1001 and 5000'
         WHEN ytd_sales <= 10000 THEN 'Between 5001 and 10000'
         ELSE 'Over 10000'
       END 'YTD Sales',
       COUNT(*) 'Number of Titles'
FROM   titles
GROUP BY CASE
         WHEN ytd_sales IS NULL THEN 'Unknown'
         WHEN ytd_sales <= 200 THEN 'Not more than 200'
         WHEN ytd_sales <= 1000 THEN 'Between 201 and 1000'
```

```
        WHEN ytd_sales <= 5000 THEN 'Between 1001 and 5000'
        WHEN ytd_sales <= 10000 THEN 'Between 5001 and 10000'
        ELSE 'Over 10000'
    END
ORDER BY MIN( ytd_sales )
```

The results are:

YTD Sales	Number of Titles
Unknown	2
Not more than 200	1
Between 201 and 1000	2
Between 1001 and 5000	9
Between 5001 and 10000	1
Over 10000	3

Next is an *UPDATE* statement that applies discounts to all of the titles. This more complicated command will discount all personal computer-related titles by 25%, all other titles by 10%, and apply only a 5% discount to titles with year-to-date sales exceeding 10,000 units. This query uses a searched *CASE* expression to perform price adjustment:

```
UPDATE titles
SET price = price *
    CASE
        WHEN ytd_sales > 10000 THEN 0.95 -- 5% discount
        WHEN type = 'popular_comp' THEN 0.75 -- 25% discount
        ELSE 0.9 -- 10% discount
    END
WHERE pub_date IS NOT NULL
```

The update has now completed three separate *UPDATE* operations in a single statement.

CAST

The *CAST* command explicitly converts an expression of one datatype to another. All vendors provide the ANSI SQL2003 syntax for *CAST*.

SQL2003 Syntax and Description

```
CAST(expression AS data_type[(length)])
```

The *CAST* function converts any expression, such as a column value or variable, into another defined datatype. The length of the datatype may be optionally supplied for those datatypes (such as *CHAR* or *VARCHAR*) that support lengths.



Be aware that some conversions, such as *DECIMAL* values to *INTEGER*, will result in rounding operations. Also, some conversion operations may result in an error if the new datatype does not have sufficient space to display the converted value.

Examples

This example retrieves the year-to-date sales as a *CHAR* and concatenates it with a literal string and a portion of the title of the book. It converts **ytd_sales** to *CHAR(5)*, plus it shortens the length of the **title** to make the results more readable.

```
SELECT CAST(ytd_sales AS CHAR(5)) + ' Copies sold of ' + CAST(title AS VARCHAR(30))
FROM titles
WHERE ytd_sales IS NOT NULL
AND ytd_sales > 10000
ORDER BY ytd_sales DESC
```

The results are:

```
-----
22246 Copies sold of The Gourmet Microwave
18722 Copies sold of You Can Combat Computer Stress
15096 Copies sold of Fifty Years in Buckingham Pala
```

4.4.3 Numeric Scalar Functions

The list of official SQL2003 numeric functions is rather small, and the different platforms provide supplementary mathematical and statistical functions. MySQL directly supports many of these functions. The other database platforms offer the same capabilities of numeric scalar functions through their own internally defined functions, but they do not share the same name as those declared by the SQL standard. The supported numeric functions and syntax are listed in [Table 4-4](#).

Table 4-4. SQL2003 numeric functions

Function	Usage
<i>ABS</i>	Returns the absolute value of a number.
<i>MOD</i>	Returns the remainder of one number divided into another.
<i>BIT_LENGTH</i>	Returns an integer value representing the number of bits in another value.
<i>CEIL</i> or <i>CEILING</i>	Rounds a noninteger value upwards to the next greatest integer. Returns an integer value unchanged.
<i>CHAR_LENGTH</i>	Returns an integer value representing the number of characters in a string expression.
<i>EXP</i>	Raises a value to the power of the mathematical constant known as <i>e</i> .
<i>EXTRACT</i>	Allows the datepart to be extracted (<i>YEAR</i> , <i>MONTH</i> , <i>DAY</i> , <i>HOUR</i> , <i>MINUTE</i> , <i>SECOND</i> , <i>TIMEZONE_HOUR</i> , or <i>TIMEZONE_MINUTE</i>) from a temporal expression.
<i>FLOOR</i>	Rounds a noninteger value downwards to the next least integer. Returns an integer value unchanged.
<i>LN</i>	Returns the natural logarithm of a number.
<i>OCTET_LENGTH</i>	Returns an integer value representing the number of octets in another value. This value is the same as <i>BIT_LENGTH/8</i> .

<i>POSITION</i>	Returns an integer value representing the starting position of a string within the search string.
<i>POWER</i>	Raises a number to a specified power.
<i>SQRT</i>	Computes the square root of a number.
<i>WIDTH_BUCKET</i>	Deposits a value into the appropriate bucket from a set of buckets covering a given range.

ABS

All platforms have standard SQL2003 support for the *ABS* function.

SQL2003 Syntax

ABS(*expression*)

ABS returns the absolute value of the number in *expression*.

Example

The following shows how to use the *ABS* function:

```
/* SQL2003 */
```

```
SELECT ABS(-1) FROM NUMBERS1
```

BIT_LENGTH, CHAR_LENGTH, and OCTET_LENGTH

All platforms stray from the ANSI standard in their support for the scalar functions for determining the length of expressions. While the platform support is nonstandard, the equivalent functionality exists under different names.

SQL2003 Syntax

The SQL2003 scalar functions for getting the length of a value take an *expression* to calculate the value and return the length as an integer. The *BIT_LENGTH* function returns the number of bits contained within the value of *expression*. The *CHAR_LENGTH* is the number of characters in the string *expression*. *OCTET_LENGTH* returns the number of octets within the string *expression*. All three of these functions will return NULL if *expression* is NULL.

BIT_LENGTH(*expression*)

CHAR_LENGTH(*expression*)

OCTET_LENGTH(*expression*)

DB2

DB2 does not support *BIT_LENGTH*, *CHAR_LENGTH*, or *OCTET_LENGTH*. DB2 does support a scalar function named *LENGTH*(), which provides similar functionality for nongraphic string types. For graphic string types, *LENGTH* returns the number of double-byte characters in a string.

MySQL

MySQL supports *CHAR_LENGTH* and the SQL2003 synonym *CHARACTER_LENGTH*().

Oracle

The closest any of the platforms get to the *BIT_LENGTH* function is Oracle. Oracle supports the *LENGTHB* function, which returns an integer value representing the number of bytes in an expression. For the length of an expression in characters, Oracle provides a *LENGTH()* function as a synonym for *CHAR_LENGTH*.

PostgreSQL

PostgreSQL supports *CHAR_LENGTH* and the SQL2003 synonym *CHARACTER_LENGTH()*.

SQL Server

SQL Server provides the *LEN* function.

Example

The following example, shown for different databases, determines the length of a string and a value retrieved from a column:

```
/* On MySQL and PostgreSQL */
```

```
SELECT CHAR_LENGTH('hello');SELECT OCTET_LENGTH(book_title) FROM titles;
```

```
/* On Microsoft SQL Server */
```

```
SELECT DATALENGTH(title) FROM titlesWHERE type = 'popular_comp'GO
```

```
/* On Oracle or DB2 */
```

```
SELECT LENGTH('HORATIO') "Length of characters"FROM dual;
```

CEIL

The *CEIL* function returns the smallest integer greater than an input value that you specify.

SQL2003 Syntax

SQL2003 supports the following two forms of the function:

```
CEIL( expression )
```

```
CEILING ( expression )
```

DB2

DB2 supports both variations of the SQL2003 syntax.

MySQL

MySQL supports only *CEILING*.

Oracle

Oracle supports only *CEIL*.

PostgreSQL

PostgreSQL has no support for either *CEIL* or *CEILING*.

SQL Server

SQL Server supports only *CEILING*.

Examples

When you pass a positive, noninteger number, the effect of *CEIL* is to round up to the next highest integer:

```
SELECT CEIL(100.1) FROM dual;
```

```
FLOOR(100.1)
```

```
-----  
      101
```

Remember, though, that with negative numbers, rounding "up" results in a lower absolute value:

```
SELECT CEIL(-100.1) FROM dual;
```

```
FLOOR(-100.1)
```

```
-----  
     -100
```

Use *FLOOR* to get behavior opposite to that of *CEIL*.

EXP

The *EXP* function returns the value of the mathematical constant *e* (approximately 2.718281) raised to the power of a specified number.

SQL2003 Syntax

All platforms support the SQL2003 syntax:

```
EXP( expression )
```

Example

The following example uses *EXP* to return an approximation of *e*:

```
SELECT EXP(1) FROM dual;
```

```
      EXP(1)  
-----  
  2.71828183
```

Use *LN* to go in the opposite direction.

EXTRACT

The *EXTRACT* function is supported by Oracle, PostgreSQL, and MySQL. The other platforms support a separate command to accomplish the same functionality.

SQL2003 Syntax

EXTRACT(date_part FROM expression)

The SQL2003 scalar function for extracting parts from a date is *EXTRACT*. The SQL2003 *EXTRACT* function takes a *date_part* and an *expression* that evaluates to a date time value.

DB2

DB2 provides unique functions that provide the same behavior as *EXTRACT*. Those functions are *DAY*, *DAYNAME*, *DAYOFWEEK*, *DAYOFWEEK_ISO*, *DAYOFYEAR*, *DAYS*, *HOURL*, *JULIAN_DAY*, *MICROSECOND*, *MIDNIGHT_SECONDS*, *MINUTE*, *MONTH*, *MONTHNAME*, *SECOND*, *TIME*, *WEEK*, *WEEK_ISO*, and *YEAR*. Look to the list shown later in this chapter in [Section 4.5.1](#) for examples on how to use these functions. As with Oracle, DB2 also offers a *TO_CHAR* function that can be used to format a string from a date value.



When extracting parts of a date value on DB2 into an integer, such as the year or minute, use the special functions provided by DB2 instead of the *TO_CHAR* function. Doing so will provide the highest performance, since a conversion from a result string into an integer won't be required.

MySQL

MySQL's implementation is extended somewhat beyond the ANSI standard. The ANSI standard does not have a provision for returning multiple fields from the same call to *EXTRACT()* (e.g., *DAY_HOUR*). The MySQL extensions try to accomplish what the combination *DATE_TRUNC()* and *DATE_PART()* do in PostgreSQL. MySQL supports the dateparts listed in [Table 4-5](#).

Table 4-5. MySQL dateparts

Type value	Meaning
<i>SECOND</i>	Seconds
<i>MINUTE</i>	Minutes
<i>HOURL</i>	Hours
<i>DAY</i>	Days
<i>MONTH</i>	Months
<i>YEAR</i>	Years
<i>MINUTE_SECOND</i>	Minutes and seconds
<i>HOURL_MINUTE</i>	Hours and minutes
<i>DAY_HOUR</i>	Days and hours
<i>YEAR_MONTH</i>	Years and months
<i>HOURL_SECOND</i>	Hours, minutes, and seconds
<i>DAY_MINUTE</i>	Days, hours, and minutes
<i>DAY_SECOND</i>	Days, hours, minutes, and seconds

Oracle

Oracle supports the SQL2003 syntax with the dateparts listed in [Table 4-6](#).

Table 4-6. Oracle dateparts

Type value	Meaning
<i>DAY</i>	The day of the month field (1-31)

<i>HOUR</i>	Returns the hour field (0-23)
<i>MINUTE</i>	The minutes field (0-59)
<i>MONTH</i>	The month field (1-12)
<i>SECOND</i>	The seconds field (0-59)
<i>TIMEZONE_HOUR</i>	The hour component of the time zone offset
<i>TIMEZONE_MINUTE</i>	The minute component of the time zone offset
<i>TIMEZONE_REGION</i>	The current time zone name
<i>TIMEZONE_ABBR</i>	The abbreviation of the current time zone
<i>YEAR</i>	The year field

PostgreSQL

PostgreSQL supports the SQL2003 syntax with a few extra dateparts. PostgreSQL supports the dateparts listed in [Table 4-7](#).

Table 4-7. PostgreSQL dateparts

Type value	Meaning
<i>CENTURY</i>	The year field divided by 100.
<i>DAY</i>	The day of the month field (1-31).
<i>DECADE</i>	The year field divided by 10.
<i>DOW</i>	Day of week (0-6, where Sunday is 0). This type only works for <i>TIMESTAMP</i> values.
<i>DOY</i>	Day of year (1-366.) The maximum returned value is only 365 for years that are not leap years. This type can only be used on <i>TIMESTAMP</i> values.
<i>EPOCH</i>	Returns the number of seconds between the epoch (1970-01-01 00:00:00-00) and the value. The result can be negative for values before the epoch.
<i>HOUR</i>	Returns the hour field (0-23).
<i>MICROSECONDS</i>	The seconds field (including fractional parts) multiplied by 1,000,000.
<i>MILLENNIUM</i>	The year field divided by 1,000.
<i>MILLISECONDS</i>	The seconds field (including fractional parts) multiplied by 1,000.
<i>MINUTE</i>	The minutes field (0-59).
<i>MONTH</i>	The month field (1-12).
<i>QUARTER</i>	The quarter of the year (1-4) that the value is in. This type can only be used with <i>TIMESTAMP</i> values.
<i>SECOND</i>	The seconds field (0-59).
<i>TIMEZONE_HOUR</i>	The hour component of the time zone offset.
<i>TIMEZONE_MINUTE</i>	The minute component of the time zone offset.
<i>WEEK</i>	The week number that the value falls on within the year.
<i>YEAR</i>	The year field.

SQL Server

SQL Server provides the function *DATEPART*(*date_part*, *expression*) as a synonym for the SQL2003 function *EXTRACT*(*date_part* FROM *expression*). SQL Server supports the dateparts listed in [Table 4-8](#).

Table 4-8. SQL Server dateparts

Type value	Meaning
------------	---------

<i>year</i>	The year field of the datetime expression. The abbreviations <i>yy</i> and <i>yyyy</i> can also be used for two-digit and four-digit years, respectively.
<i>quarter</i>	The quarter that the datetime expression falls on. The abbreviations <i>q</i> and <i>qq</i> can also be used.
<i>dayofyear</i>	The day of the year for the date time expression. The abbreviations <i>y</i> and <i>dy</i> can also be used.
<i>day</i>	The day of the month for the date time expression. The abbreviations <i>d</i> and <i>dd</i> can also be used.
<i>week</i>	The week of the year for the date time expression. The abbreviations <i>wk</i> and <i>ww</i> can also be used.
<i>weekday</i>	The day of the week for the date time expression. The abbreviation <i>dw</i> can also be used.
<i>hour</i>	The hour of the day for the date time expression. The abbreviation <i>hh</i> can also be used.
<i>minute</i>	The minute of the hour for the date time expression. The abbreviations <i>n</i> and <i>mi</i> can also be used.
<i>second</i>	The second of the minute for the date time expression. The abbreviations <i>s</i> and <i>ss</i> can also be used.
<i>millisecond</i>	The millisecond for the date time expression. The abbreviation <i>ms</i> can also be used.

Example

This example extracts dateparts from several date time values:

```
/* On MySQL */
```

```
SELECT EXTRACT(YEAR FROM "2013-07-02");2013
```

```
SELECT EXTRACT(YEAR_MONTH FROM "2013-07-02 01:02:03");201307
```

```
SELECT EXTRACT(DAY_MINUTE FROM "2013-07-02 01:02:03");20102
```

```
/* On PostgreSQL */
```

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
```

20

FLOOR

The *FLOOR* function returns the largest integer less than an input value that you specify.

SQL2003 Syntax

All platforms support the SQL2003 syntax:

```
FLOOR( expression )
```

Examples

When you pass a positive number, the effect of *FLOOR* is to eliminate anything after the decimal point:

```
SELECT FLOOR(100.1) FROM dual;
```

```
FLOOR(100.1)
```

```
-----
```

```
100
```

Remember, though, that with negative numbers, going in the "less than" direction corresponds to increasingly larger absolute values:

```
SELECT FLOOR(-100.1) FROM dual;
```

```
FLOOR(-100.1)
```

```
-----
```

```
-101
```

Use *CEIL* to get behavior opposite to *FLOOR*.

LN

The *LN* function returns the natural logarithm of a number, which is the power to which you would need to raise the mathematical constant *e* (approximately 2.718281) in order to get the number in question as the result.

SQL2003 Syntax

```
LN( expression )
```

DB2, Oracle, PostgreSQL

DB2, Oracle, and PostgreSQL support the SQL2003 syntax for the *LN* function. DB2 and PostgreSQL also support the use of *LOG* as a synonym for *LN*.

MySQL and SQL Server

MySQL and SQL Server call their natural logarithm function *LOG*:

```
LOG( expression )
```

Example

The following, Oracle-based example shows the natural logarithm of a number closely approximating the mathematical constant known as *e*:

```
SELECT LN(2.718281) FROM dual;
```

```
LN(2.718281)
```

```
-----
```

```
.999999695
```

Use the *EXP* function to go in the other direction.

MOD

The *MOD* function returns the remainder of a dividend divided by a divider. All platforms support the SQL2003 syntax for the *MOD* function.

SQL2003 Syntax

```
MOD( dividend, divider )
```

The standard syntax for the *MOD* function is to return the remainder of dividend divided by the divider; returns the dividend if the divider is 0.

Example

The following shows how to use the *MOD* function from within a *SELECT* statement:

```
SELECT MOD(12, 5) FROM NUMBERS
```

```
2
```

POSITION

The *POSITION* function returns an integer that indicates the starting position of a string within the search string.

SQL2003 Syntax

```
POSITION( string1 IN string2 )
```

The standard syntax for the *POSITION* function is to return the first location of *string1* within *string2*. *POSITION* returns 0 if *string1* does not occur within *string2* and NULL if either argument is NULL.

DB2

DB2's equivalent function is *POSSTR*.

MySQL

MySQL supports *POSITION* as defined in SQL2003.

Oracle

Oracle's equivalent function is called *INSTR*.

PostgreSQL

PostgreSQL supports *POSITION* as defined in SQL2003.

SQL Server

SQL Server has both *CHARINDEX* and *PATINDEX*. *CHARINDEX* and *PATINDEX* are very similar, except that *PATINDEX* allows the use of wildcard characters in the search criteria.

Examples

```
/* DB2 */
```

```
SELECT POSSTR('bar', 'foobar');
```

```
4
```

```
/* On MySQL */
```

```
SELECT LOCATE('bar', 'foobar');4
```

/ On MySQL and PostgreSQL */*

```
SELECT POSITION('fu' IN 'snafhu');0
```

/ On Microsoft SQL Server */*

```
SELECT CHARINDEX( 'de', 'abcdefg' )GO4
```

```
SELECT PATINDEX( '%fg', 'abcdefg' )GO6
```

POWER

Use *POWER* to raise a number to a specific value.

SQL2003 Syntax

```
POWER( base, exponent )
```

The result of the *POWER* function is *base* raised to the *exponent* power, or *base^{exponent}*. If *base* is negative, then *exponent* must be an integer.

DB2, Oracle, PostgreSQL, and SQL Server

These vendors all support the SQL2003 syntax.

MySql

MySQL supports the same functionality, but requires the use of the keyword *POW*:

```
POW( base, exponent )
```

Example

Raising a positive number to an exponent is straightforward:

```
SELECT POWER(10,3) FROM dual;
```

```
POWER(10,3)
```

```
-----
```

```
1000
```

Anything raised to the 0th power evaluates to 1:

```
SELECT POWER(0,0) FROM dual;
```

```
POWER(0,0)
```

```
-----
```

```
1
```

Negative exponents move the decimal point to the left:

```
SELECT POWER(10,-3) FROM dual;
```

```
POWER(10,-3)
```

```
-----
```

```
.001
```

SQRT

The *SQRT* function returns the square root of a number.

SQL2003 Syntax

All platforms support the SQL2003 syntax:

```
SQRT( expression )
```

Example

```
SELECT SQRT(100) FROM dual;
```

```
SQRT(100)
```

```
-----
```

```
10
```

WIDTH_BUCKET

The *WIDTH_BUCKET* function assigns values to buckets in an equiwidth histogram.

SQL2003 Syntax

In the following syntax, *expression* represents a value to be assigned to a bucket. You would typically base *expression* on one or more columns returned by a query.

```
WIDTH_BUCKET(expression, min, max, buckets)
```

The *buckets* argument specifies the number of buckets to create over the range defined by *min* through *max*. *min* is inclusive, whereas *max* is not. The value from *expression* is assigned to one of those buckets, and the function then returns the corresponding bucket number. When *expression* falls outside the range of buckets, the function returns either 0 or *max*+1, depending on whether *expression* is lower than *min* or greater than or equal to *max*.

Examples

The following example divides the integer values 1 through 10 into two buckets:

```
SELECT x, WIDTH_BUCKET(x,1,10,2)
```

```
FROM pivot;
```

X WIDTH_BUCKET(X,1,10,2)

1	1
2	1
3	1
4	1
5	1
6	2
7	2
8	2
9	2
10	3

The following example is more interesting. It divides 11 values from 1 through 10 into three buckets, and illustrates the distinction between *min* being inclusive and *max* being noninclusive:

SELECT x, WIDTH_BUCKET(x,1,10,3)

FROM pivot;

X WIDTH_BUCKET(X,1,10,3)

1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	3
9.9	3
10	4

Pay particular attention to the results for X=1, 9.9, and 10. An input value of *min*, 1 in this example, falls into that first bucket, proving that the lower end of the range for bucket #1 is defined as $x \geq \text{min}$. An input value of *max*, however, falls *outside* the highest bucket. In this example, 10 falls into the overflow bucket numbered $\text{max}+1$. The value 9.9 falls into bucket #3, illustrating that the upper end of the range for the highest bucket is defined as $x < \text{max}$.

4.4.4 String Functions and Operators

Basic string functions and operators offer a number of capabilities and return a string value as a result. Some string functions are dyadic, indicating that they operate on two strings at once. SQL2003 supports the string functions listed in [Table 4-9](#).

Table 4-9. SQL String functions and operators

Function or operator	Usage
Concatenation operator	Appends two or more literal string expressions, column values, or variables together into one string.
<i>CONVERT</i>	Converts a string to a different representation within the same character set.
<i>LOWER</i>	Converts a string to all lowercase characters.
<i>OVERLAY</i>	Returns the result of replacing a substring of one string with another.
<i>SUBSTRING</i>	Returns a portion of a string.
<i>TRANSLATE</i>	Converts a string from one character set to another.
<i>TRIM</i>	Removes leading characters, trailing characters, or both from a character string.
<i>UPPER</i>	Converts a string to all uppercase characters.

Concatenation Operator

SQL2003 defines a concatenation operator (||), which joins two distinct strings into one string value.

DB2

DB2 supports the SQL2003 concatenation operator and the *CONCAT* function as a synonym.

MySQL

MySQL supports *CONCAT*() as a synonym for the SQL2003 concatenation operator.

Oracle and PostgreSQL

PostgreSQL and Oracle support the SQL2003 double vertical bar concatenation operator.

SQL Server

SQL Server uses the plus sign (+) as a synonym for the SQL2003 concatenation operator. SQL Server has the system setting *CONCAT_NULL_YIELDS_NULL*, which can be set to alter the behavior when NULL values are used in the concatenation of string values.

Examples

```
/* SQL2003 Syntax */
```

```
'string1' || 'string2' || 'string3'
```

```
'string1string2string3'
```

```
/* On MySQL */
```

```
CONCAT('string1', 'string2')
```

```
'string1string2'
```

If any of the concatenation values are NULL, the entire returned string is NULL. Also, if a numeric value is concatenated, it is implicitly converted to a character string.

```
SELECT CONCAT('My ', 'bologna ', 'has ', 'a ', 'first ', 'name...');
```

```
'My bologna has a first name...'
```

```
SELECT CONCAT('My ', NULL, 'has ', 'first ', 'name...');NULL
```

CONVERT and TRANSLATE

The *CONVERT* function alters the representation of a character string within its character set and collation. For example, *CONVERT* might be used to alter the number of bits per character.

TRANSLATE alters the character set of a string value from one base-character set to another. Thus, *TRANSLATE* might be used to translate a value from the English character set to a Kanji (Japanese) or Cyrillic (Russian) character set. The translation must already exist, either by default or by having been created using the *CREATE TRANSLATION* command.

SQL2003 Syntax

CONVERT (*char_value* USING *conversion_char_name*)

TRANSLATE(*char_value* USING *translation_name*)

CONVERT converts *char_value* to the character set with the name supplied in *conversion_char_name*. *TRANSLATE* converts *char_value* to the character set provided in *translation_name*.

Among the platforms, only Oracle supports *CONVERT* and *TRANSLATE* with the same meaning as SQL2003. Oracle's implementation of *TRANSLATE* is very similar to SQL2003, but not identical. In its implementation, Oracle accepts only two arguments and translates only between the database character set and the national language support character set.

MySQL's implementation of the *CONV* function only translates numbers from one base to another. In contrast, SQL Server's implementation of *CONVERT* is a very rich utility that alters the base datatype of an expression, but is otherwise dissimilar to the SQL2003 *CONVERT* function. PostgreSQL does not support *CONVERT*, and its implementation of *TRANSLATE* morphs any occurrence of a character string to any other character string.

DB2

DB2 *does not* support *CONVERT*, and its support for the *TRANSLATE* function is not ANSI standard. *TRANSLATE* is used to translate substrings and has historically acted as a synonym for *UPPER*, since *UPPER* was only recently added to DB2. When *TRANSLATE* is used with only a string expression as an argument, the result will be an uppercase equivalent of that argument. When used with more than one argument, as in *TRANSLATE(source, replacements, match)*, the function translates all characters in *source* that are also in *match*. Each character in *source* will be replaced with the character in *replacements* that is found in the same position as the matching character in *match*. Following are some examples:

TRANSLATE('Hello, World!')

'HELLO, WORLD!'

TRANSLATE('Hello, World!', 'wZ', 'IW')

'Hewwo, Zorwd!'

MySQL

MySQL *does not* support *TRANSLATE* or *CONVERT*.

Oracle

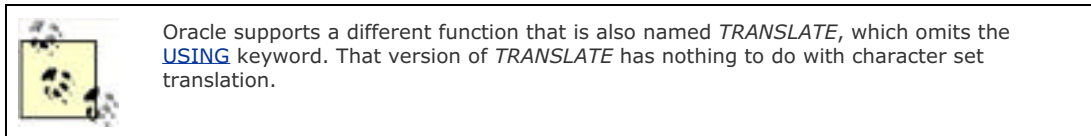
Oracle supports the following *CONVERT* and *TRANSLATE* functions:

CONVERT(*char_value*, *target_char_set*, *source_char_set*)

TRANSLATE(*char_value* USING {*CHAR_CS* | *NCHAR_CS*})

Under Oracle's implementation, the *CONVERT* function returns the text of *char_value* in the target character set. The *char_value* is the string to convert, *target_char_set* is the name of the character set into which the string is to be converted, and *source_char_set* is the name of the character set in which *char_value* was originally stored.

Oracle's *TRANSLATE* function follows the ANSI syntax, but you can choose between two character sets only: the database character set (*CHAR_CS*) and the national character set (*NCHAR_CS*).



Both the target and source character set names can be passed either as literal strings, in variables, or in columns from a table. Note that replacement characters might be substituted when converting from or to a character set that does not support a representation of all the characters used in the conversion.

Oracle supports several common character sets, including *US7ASCII*, *WE8DECDEC*, *WE8HP*, *F7DEC*, *WE8EBCDIC500*, *WE8PC850*, and *WE8ISO8859P1*. For example:

```
SELECT CONVERT('Gro2', 'US7ASCII', 'WE8HP')FROM DUAL;Gross
```

PostgreSQL

PostgreSQL supports the ANSI standard *CONVERT* and conversions can be defined by using *CREATE CONVERSION*. PostgreSQL's implementation of the *TRANSLATE* function offers a large superset of functions that can convert any occurrence of one text string to another within another specified string.

```
TRANSLATE (character_string, from_text, to_text)
```

Here are some examples:

```
SELECT TRANSLATE('12345abcde', '5a', 'XX');1234XXbcde'
```

```
SELECT TRANSLATE(title, 'Computer', 'PC')FROM titlesWHERE type = 'Personal_computer'
```

```
SELECT CONVERT('PostgreSQL' USING iso_8859_1_to_utf_8)
```

```
'PostgreSQL'
```

SQL Server

SQL Server *does not* support *TRANSLATE*. SQL Server's implementation of the *CONVERT* function does not follow the SQL2003 specification. Instead, it is functionally equivalent to the *CAST* function.

```
CONVERT (data_type[(length) | (precision,scale)], expression[,style])
```

The *style* clause is used to define the format of a date conversion. Refer to the SQL Server documentation for more information. Following is an example:

```
SELECT title, CONVERT(char(7), ytd_sales)
```

```
FROM titles
```

```
ORDER BY title
```

```
GO
```

LOWER and UPPER

The functions *LOWER* and *UPPER* allow the case of a string to be altered quickly and easily, so that all the characters are lower- or uppercase, respectively. These functions are supported in all the database implementations covered in this book. The different database platforms also support a variety of other text formatting functions that are specific to their implementation.

SQL2003 Syntax

LOWER(string)

UPPER(string)

LOWER converts *string* into a lowercase string. *UPPER* is the uppercase counterpart of *LOWER*.

DB2 and MySQL

These platforms support the SQL2003 *UPPER* and *LOWER* scalar functions, as well as the synonyms *UCASE* and *LCASE*.

Oracle, PostgreSQL, and SQL Server

These platforms support the SQL2003 *UPPER* and *LOWER* scalar functions, as in the following example:

Example

```
SELECT LOWER('You Talkin To ME?'), UPPER('you talking to me?!');
```

```
you talkin to me?, YOU TALKING TO ME?!
```

OVERLAY

The *OVERLAY* function embeds one string into another and returns the result.

SQL2003 Syntax

OVERLAY(string PLACING embedded_string FROM start[FOR length])

If any of the inputs are NULL, the *OVERLAY* function returns a NULL. The *embedded_string* replaces the *length* characters in *string* starting at character position *start*. If the *length* is not specified, then the *embedded_string* will replace all characters after *start* in *string*.

DB2, MySQL, Oracle, and SQL Server

These platforms do not have support for the *OVERLAY* function. You can simulate the *OVERLAY* function on these platforms by using a combination of *SUBSTRING* and the concatenation operator.

PostgreSQL

PostgreSQL supports the ANSI standard for *OVERLAY*.

Examples

This is an example of how to use the *OVERLAY* function:

```
/* SQL2003 and PostgreSQL */
```

```
SELECT OVERLAY('DONALD DUCK' PLACING 'TRUMP' FROM 8) FROM NAMES;'DONALD TRUMP'
```

SUBSTRING

The *SUBSTRING* function allows one character string to be returned from another.

SQL2003 Syntax

```
SUBSTRING(extraction_string FROM starting_position [FOR length]
```

```
[COLLATE collation_name])
```

If any of the inputs are NULL, the *SUBSTRING* function returns a NULL. The *extraction_string* is the source where the character value is to be extracted. It may be a literal string, a column in a table with a character datatype, or a variable with a character datatype. The *starting_position* is an integer value telling the function at which position to begin performing the extract. The optional *length* is an integer value that tells the function how many characters to extract, starting at the *starting_position*. If the optional *FOR* is omitted, then the substring starting at *starting_position* and continuing to the end of the *extraction_string* is returned.

DB2

```
SUBSTR(extraction_string, starting_position [, length])
```

DB2's implementation, *SUBSTR*, largely functions the same way as SQL2003's *SUBSTRING*. It does not support the *COLLATE* clause. If *length* is omitted, the remainder of the string (starting at *starting_position*) is returned.

MySQL

```
SUBSTRING(extraction_string FROM starting_position)
```

MySQL's implementation assumes that the characters are to be extracted from the starting position and will continue to the end of the character string.

Oracle

```
SUBSTR(extraction_string, starting_position [, length])
```

Oracle's implementation, *SUBSTR*, largely functions the same way as SQL2003's *SUBSTRING*. It *does not* support the *COLLATE* clause. When a *starting_position* is a negative number, Oracle counts from the end of the *extraction_string*. If *length* is omitted, the remainder of the string (starting at *starting_position*) is returned.

PostgreSQL

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

PostgreSQL largely supports the ANSI standard, except that it does not accept the *COLLATE* clause.

SQL Server

`SUBSTRING(extraction_string [FROM starting_position] [FOR length])`

SQL Server largely supports the ANSI standard, except that it does not allow the *COLLATE* clause. SQL Server allows this command to be applied to text, image, and binary datatypes; however, the *starting_position* and *length* represent the number of bytes rather than the number of characters to count.

Examples

These examples generally work on any one of the five database platforms profiled in this book. Only the second Oracle example, with a negative starting position, fails on the others (assuming, of course, that Oracle's *SUBSTR* is translated into *SUBSTRING*):

/ On Oracle, counting from the left */*

```
SELECT SUBSTR('ABCDEFGF',3,4) FROM DUAL;'CDEF'
```

/ On Oracle, counting from the right */*

```
SELECT SUBSTR('ABCDEFGF',-5,4) FROM DUAL;'CDEF'
```

/ On MySQL */*

```
SELECT SUBSTRING('Be vewy, vewy quiet' FROM 5);'wy, vewy quiet'
```

/ On PostgreSQL or SQL Server */*

```
SELECT au_iname, SUBSTRING(au_fname, 1, 1)FROM authorsWHERE au_iname = 'Carson'
```

Carson C

TRIM

The *TRIM* function removes leading characters, trailing characters, or both from a specified character string or *BLOB* value. This function also removes other types of characters from a specified character string. The default function is to trim the specified character from both sides of the character string. If no removal character is specified, *TRIM* removes spaces by default.

SQL2003 Syntax

```
TRIM( [ [ {LEADING | TRAILING | BOTH} ] [removal_char] FROM ]
```

```
target_string
```

```
[COLLATE collation_name])
```

The *removal_char* is the character to be stripped out. The *target_string* is the character string from which characters are to be stripped. If a *removal_char* is not specified, then *TRIM* strips out spaces. The *COLLATE* clause forces the result set of the function into another pre-existing collation set.

DB2

DB2 provides the functions *LTRIM* and *RTRIM* to trim off leading spaces or trailing spaces, respectively.

MySQL, Oracle, and PostgreSQL

These platforms support the SQL2003 syntax of *TRIM*.

Microsoft SQL Server

SQL Server provides the functions *LTRIM* and *RTRIM* to trim off leading spaces or trailing spaces, respectively. On SQL Server the *LTRIM* and *RTRIM* cannot be used to trim other types of characters.

Examples

```
SELECT TRIM(' wamalamadingdong ');wamalamadingdong'
```

```
SELECT LTRIM( RTRIM(' wamalamadingdong '));wamalamadingdong'
```

```
SELECT TRIM(LEADING '19' FROM '1976 AMC GREMLIN');76 AMC GREMLIN'
```

```
SELECT TRIM(BOTH 'x' FROM 'xxxWHISKEYxxx');WHISKEY'
```

```
SELECT TRIM(TRAILING 'snack' FROM 'scooby snack');scooby '
```

4.5 Platform-Specific Extensions

The following sections provide a full listing and description of each vendor-supported function. The functions are platform-specific. Thus, a MySQL function, for example, is not guaranteed to be supported by any other vendor.

4.5.1 DB2-Supported Extensions

This section provides an alphabetical listing of DB2-supported functions that are specific to DB2 and not from the ANSI standard, with examples and corresponding results.

ABSVAL(number)

Synonym for *ABS(number)*. For example:

```
VALUES( ABSVAL( -1 ) ) -> 1
```

ACOS(number)

Returns the arc cosine of a number ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

ASCII(text)

Returns the ASCII code of the first character of text. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(number, nbr)

Returns the arctangent of *number* and *nbr*. The values for *number* and *nbr* are not restricted, but the results range from $-\pi$ to π and are expressed in radians. *ATAN2(x,y)* is similar to *ATAN(y/x)* with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
VALUES( ATAN2( 3.1415, 1 ) ) -> +3.08177595443792E-001
```

ATANH(number)

Returns the hyperbolic arctangent of *number*. The values for *number* are not restricted, but the results range

from - to and are expressed in radians. For example:

VALUES(ATANH(0)) -> +0.00000000000000E+000

BIGINT(expression)

Returns a 64-bit integer representation of *expression*, where *expression* can be a number, character string, date, time, or timestamp. For example:

SELECT BIGINT('1991-11-22') -> 19911122

BLOB(string [,length])

Returns a *BLOB* representation of *string*, where *string* can be a string representation of any type. This example will return *BLOBs* from the **employee_images** table that contain the binary string "JFIF."

SELECT image FROM employee_images WHERE image LIKE BLOB('%JFIF%')

CHAR(expression [,length])

Returns a fixed-length character string representation of *expression*, where the optional parameter *length* is the length attribute of the character value returned.

CHR(number)

Returns the character having the ASCII code value equal to *number*. For example:

VALUES(CHR(120)) -> 'x'

CLOB(string [,length])

Returns a *CLOB* representation of *string*.

COALESCE(expression [,...])

Returns the first argument that is not NULL. For example:

SELECT COALESCE(1,2,3) -> 1

CONCAT(string1, string2)

Returns *string1* concatenated with *string2*. It is equivalent to the concatenation operator (||). For example:

SELECT CONCAT(au_lname, au_fname) FROM authors -> 'JeffersonThomas'

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

COSH(number)

Returns the hyperbolic cosine of *number*. For example:

VALUES(COSH(3.1415)) -> +1.15908832931176E+001

COT(number)

Returns the cotangent of *number*. For example:

SELECT COT(3.1415) -> -10792.88993953

DATE(expression)

Returns a date value from *expression*. If *expression* is an integer value, then the returned date is equivalent to *expression-1* days after January 1, 0001. For example:

SELECT DATE(3) -> '01/03/0001'

DAY(expression)

Returns the day number in a date expression. For example:

SELECT DAY('1999-04-15') -> 15

DAYNAME(expression)

Returns the name of the day in *expression* using the locale of the database server. For example:

SELECT DAYNAME('1999-04-15') -> 'Thursday'

DAYOFWEEK(expression), DAYOFWEEK_ISO(expression), DAYOFYEAR(expression)

Returns the day of the week or year in *expression*. *DAYOFWEEK* returns an integer value in the range 1-7, where 1 represents Sunday. *DAYOFWEEK_ISO* has the same range as *DAYOFWEEK*, but the value of 1 represents Monday. *DAYOFYEAR* returns an integer value in the range 1-366, where the value returned is the day of the year starting with 1 for January 1st. For example:

VALUES(DAYOFWEEK('1999-04-15')) -> 5

VALUES(DAYOFWEEK_ISO('1999-04-15')) -> 4

VALUES(DAYOFYEAR('1999-04-15')) -> 105

DAYS(expression)

Returns one more than the number of days between *expression* and January 1, 0001. For example:

VALUES(DAYS('1999-04-15')) -> 729859

DBCLOB(expression [,length])

Returns a *DBCLOB* representation of a graphic string type, where *length* specifies the length of the *DBCLOB* value returned.

DBPARTITIONNUM(column)

Returns the database partition number of the row containing *column*. For more information, please look to the DB2 user documentation for *DBPARTITIONNUM*.

DECIMAL(expression [,precision [,scale [,decimal_char]]], DEC(expression [,precision [,scale [,decimal_char]]]

Returns a decimal representation of *expression*. The optional arguments control the *precision* and *scale* used in the result, whereas *decimal_char* is the decimal character found in *expression*, if any.

DECRYPT_BIN(*data* [, *password*]) or *DECRYPT_CHAR*(*data* [, *password*])

Returns decrypted data using the optional *password*. The result of *DECRYPT_BIN* is a *VARCHAR FOR BIT DATA*, whereas *DECRYPT_CHAR* returns a *VARCHAR*.

DEGREES(*number*)

Returns the number of degrees converted from the argument expressed in radians. For example:

VALUES(DEGREES(3.1415926)) -> +1.79999996929531E+002

DEREF(*expression*)

Returns the object reference of *expression*, where *expression* must return a *REF* to an object. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

DIFFERENCE(*expression1*, *expression2*)

Returns the difference between the sounds of two strings based on their *SOUNDEX* values. The result is an integer in the range of 0-4, where 4 is the best *SOUNDEX* match. For example:

VALUES(DIFFERENCE('thimble', 'nimble')) -> 4

DIGITS(*number*)

Returns the *number* argument into a character string of digits. The result of *DIGITS* does not contain decimal or sign characters and it may be padded on the left with zeros. For example:

VALUES(DIGITS(DEC('-3.1415926', 12, 8))) -> 000031415926

DOUBLE(*number*)

Returns *number* converted to a *DOUBLE*.

ENCRYPT(*data* [, *password* [, *hint*]])

Returns *data* encrypted with the optional *password*. The optional *hint* argument allows a user to store up to a 32-byte password hint that is encoded into the result value. To get the password hint, look to the *GETHINT* function. For example:

UPDATE employee SET ssn = ENCRYPT(ssn, 'luvbug', 'Herbie is-a?')

WHERE empid = '54321-AD'

EVENT_MON_STATE(*expression*)

Returns the state of an event monitor. For details on usage or available event monitors, please look to the DB2 user documentation.

FLOAT(*number*)

FLOAT is a synonym for *DOUBLE*.

GETHINT(*data*)

Returns the password hint encoded in *data*. The *data* argument must be encrypted with the *ENCRYPT* function. For example:

SELECT GETHINT(ssn) FROM EMPLOYEE WHERE empid = '54321-AD'

'Herbie is-a?'

GENERATE_UNIQUE()

Returns a 13-byte *CHAR(13) FOR BIT DATA* value that is guaranteed to be unique from other invocations within the same database. For example:

INSERT INTO employee(id, emp_name)

VALUES(GENERATE_UNIQUE(), 'Bob Smith')

GRAPHIC(expression [,length])

Returns a fixed-length graphic string representation of *expression*, where the optional *length* argument is the length of the result.

HASHEDVALUE(column)

Returns the partitioning map index of the row containing *column*. For more information, please look to the DB2 user documentation for *HASHEDVALUE()*.

HEX(expression)

Returns a hexadecimal representation of *expression*. For example:

VALUES(HEX(255)) -> 000000FF

HOUR(expression)

Returns the hour of part of the time value in *expression*. For example:

SELECT HOUR(execution_time), COUNT(*) FROM trades

GROUP BY HOUR(execution_time)

```
1      2
-----
      8      2058
      9      856
     10      912
     11      714
...

```

IDENTITY_VAL_LOCAL()

Returns the value most recently assigned to an identity column. Returns NULL when an identity column has not been inserted into a table since the last *COMMIT* or *ROLLBACK* statement.

INSERT(expression1, expression2, expression3, expression4)

Returns *expression1* with *expression4*, replacing *expression3* bytes at position *expression2*. For example:

VALUES (INSERT('food', 2, 2, 'are')) -> 'fared'

INTEGER(expression)

Returns an integer representation of *expression*.

JULIAN_DAY(expression)

Returns the Julian day number for the date value in *expression*. The value returned is the number of days between the date value in *expression* and the start of the Julian date calendar, January 1, 4713 B.C. For example:

VALUES(JULIAN_DAY('1999-04-15')) -> 2451284

LEFT(string, length)

Returns the first *length* bytes from *string*. For example:

VALUES(LEFT('Hello, World!', 5)) -> 'Hello'

LENGTH(expression)

Returns the integer length of *expression*, or NULL if expression is NULL. For example:

VALUES(LENGTH('Hello, World!')) -> 13

LOCATE(substring, string [,starting_pos])

Returns the location of *substring* within *string* or zero, if the substring is not found within *string*. The optional *starting_pos* can be used to specify the starting position within *string* to begin the search. For example:

VALUES(LOCATE('World', 'Hello, World!')) -> 8

LOG(number)

Returns the natural logarithm of *number*, the same as *LN(number)*.

LOG10(number)

Returns the base 10 logarithm of *number*. For example:

VALUES(LOG10(50)) -> +1.69897000433602E+000

LONG_VARCHAR(string)

Returns a *LONG_VARCHAR* representation of *string*.

LONG_VARGRAPHIC(string)

Returns a *LONG_VARGRAPHIC* representation of *string*.

LTRIM(string)

Removes all characters inset from the left of *string*. For example:

VALUES(LTRIM(' Howdy! ')) -> 'Howdy! '

MICROSECOND(expression)

Returns the microsecond part of the time value in *expression*. For example:

VALUES(MICROSECOND(CURRENT_TIMESTAMP)) -> 252270

MIDNIGHT_SECONDS(expression)

Returns the number of seconds that have elapsed between midnight and the time value in *expression*. For example:

VALUES(MIDNIGHT_SECONDS('08:20:15')) -> 30015

MINUTE(expression)

Returns the minute part of the time value in *expression*. For example:

VALUES(MINUTE('08:20:15')) -> 20

MONTH(expression)

Returns the month part of the date value in *expression*. For example:

VALUES(MONTH('1999-04-15')) -> 4

MONTHNAME(expression)

Returns the month's name for the date value stored in *expression* using the current locale of the database. For example:

VALUES(MONTHNAME('1999-04-15')) -> 'April'

MULTIPLY_ALT(number1, number2)

Returns the product of *number1* and *number2*. The function *MULTIPLY_ALT* is a good alternative to the standard multiplication operator (*) when the numbers being multiplied have a precision greater than 31. For example:

VALUES(MULTIPLY_ALT(DECIMAL('256'), DECIMAL('256'))) -> 65536

NULLIF(expression1, expression2)

Returns NULL if *expression1* and *expression2* are equal; otherwise the function returns *expression1*. For example:

VALUES(NULLIF('1999-04-15', '2000-04-15')) -> '1999-04-15'

POSSTR(source, search)

Returns the character position of the first occurrence of *search* within *source* where the first character position is 1. For example:

VALUES(POSSTR('Hello, World!', 'World')) -> 8

QUARTER(expression)

Returns an integer ranging from 1 to 4, representing the quarter of the year that contains the date in *expression*. For example:

VALUES(QUARTER('2004-04-15')) -> 2

RADIANS(expression)

Returns the number of radians from *expression*, which is expressed in degrees. For example:

VALUES(RADIANS(180)) -> +3.14159265358979E+000

RAISE_ERROR(sqlstate, errorstring)

Raises an error upon execution. This function is useful for generating critical errors in complicated SQL statements and stored procedures. The *sqlstate* parameter is used for passing error code information back to the application executing the SQL statement, and the *errorstring* parameter is used for a custom error message. For example:

```
SELECT a.au_fname, a.au_name,  
  
CASE WHEN t.ytd_sales = 0 THEN 'no sales'  
      WHEN t.ytd_sales > 0 THEN 'OK'  
      ELSE RAISE_ERROR('70001', 'Sales should not be negative.')  
END  
  
FROM authors a, titleauthor, titles t  
  
WHERE titleauthor.au_id = a.au_id AND  
      Titleauthor.title_id = t.title_id  
  
GROUP BY a.au_fname, a.au_name
```

RAND([seed])

Returns a random floating-point value between 0 and 1, using *seed* as the random number generator seed.

REAL(number)

Returns a single-precision floating point representation of *number*.

REC2XML(decimal, format, rowtags, column_name [, . . .])

Returns an XML string containing XML tags, as well as column names and column data. The arguments are described in the following list:

decimal

Decimal value greater than 0, but less than or equal to 6.0, that contains the factor to expand string values to compensate for the string value expanding due to character replacements for the XML values and elements.

format

Case-sensitive string value equal to "COLATTVAL" or "COLATTVAL_XML". If column values can contain XML data that would need to be translated to produce valid XML output, then use "COLATTVAL". If the columns will not contain special XML markup characters, then use "COLATTVAL_XML". Column names containing special markup characters will still be translated (i.e., "escaped") when "COLATTVAL_XML" is chosen.

rowtags

String value to name the row element containing the column data. If *rowtags* is an empty string, then the value of **row** is used. If *rowtags* contains a string of only spaces, then the enclosing row element will be omitted in the output.

column_name

The name of a column to place into the XML result. For example:

```
SELECT REC2XML(1.3, 'COLATTVAL', 'Author', au_id, au_fname,  
    au_lname)  
FROM AUTHORS WHERE au_id = '172-32-1176'  
  
<Author>  
    <column name="AU_ID">172-32-1176</column>  
    <column name="AU_FNAME">Johnson</column>  
    <column name="AU_LNAME">White</column>  
</Author>
```

REPEAT(string, number)

Returns a character string that is composed of *string* repeated *number* times. For example:

```
VALUES( CHAR(REPEAT('Duck ', 3)) )    -> 'Duck Duck Duck'
```

REPLACE(string, search_string, replacement_string)

Returns *string* with every occurrence of *search_string* replaced with *replacement_string*. For example:

```
VALUES( REPLACE('change', 'e', 'ing') ) -> 'changing'
```

RIGHT(string, number)

Returns the *number* rightmost bytes from *string*. For example:

```
VALUES( RIGHT('Hello, World!', 6) )    -> 'World!'
```

ROUND(number[, decimal])

Returns *number* rounded to *decimal* places right of the decimal point. When *decimal* is omitted, *number* is rounded to places. Note that *decimal*, an integer, can be negative to round off digits left of the decimal point. For example:

```
VALUES( ROUND(12345.6789, 2) ) -> 12345.6800
```

RTRIM(string)

Returns *string* with all trailing whitespace characters removed. For example:

```
VALUES( RTRIM(' welcome ') )    -> ' welcome'
```

SECOND(expression)

Returns the seconds part of the time value in *expression*. For example:

```
VALUES( SECOND('08:20:15') ) -> 15
```

SIGN(number)

When *number* < 0, returns -1. When *number* = 0, returns 0. When *number* > 0, returns 1. For example:

VALUES(SIGN(-3.1415926), SIGN(0), SIGN(3.1415926))

-1.0000000000000000E+000 0 +1.0000000000000000E+000

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SINH(number)

Returns the hyperbolic sine of *number*.

SMALLINT(number)

Returns a *SMALLINT* value equivalent to *number*.

SOUNDEX(string)

Returns a character string containing the phonetic representation of *string*. This function allows words that are spelled differently but sound alike in English to be compared for equality. For example:

VALUES(SOUNDEX('thimble')) -> 'T514'

SPACE(number)

Returns a string composed of *number* spaces. For example:

VALUES(SPACE(5)) -> ' '

STDDEV({ ALL | DISTINCT } expression)

Returns the standard deviation of the values contained within *expression*. The *ALL* keyword specifies the default behavior in which all values in *expression* are used in the standard deviation. The keyword *DISTINCT* omits duplicate values in *expression* when doing the calculation. For example:

SELECT STDDEV(values) FROM NUMBERS -> 0.0642

SUBSTR(string, start [, length])

Returns a substring of *string* of *length* bytes starting at *start*. If *length* is too large to permit a valid substring, then *string* is padded with trailing spaces until a valid substring can be found. The starting position, *start*, must be an integer value in the range of 1 to the length of *string*. For example:

VALUES(SUBSTR('Hello, World!', 8, 5)) -> 'World'

TABLE_NAME(object [,schema])

Returns the unqualified name of *object* after resolving any aliases. The optional *schema* argument specifies the database schema to use in resolving the object name. For example:

VALUES(TABLE_NAME('trades')) -> 'trades'

TABLE_SCHEMA(object [,schema])

Returns the schema name of the *object* after resolving any aliases. The optional *schema* argument specifies the database schema to use in resolving the object name. For example:

VALUES(TABLE_SCHEMA('trades')) -> 'MYSHEMA'

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

SELECT TAN(3.1415) -> -0.000093

TANH(number)

Returns the hyperbolic tangent of *number*.

TIME(expression)

Returns a time value equivalent to the time in *expression*. For example:

VALUES(TIME('2003-04-15 08:20:15')) -> '08:20:15'

TIMESTAMP(expression1 [,expression2])

Returns a timestamp value created from date and time values in *expression1* and *expression2*. If only *expression1* is given, then it must contain a valid date and time value. If both arguments are given, then *expression1* must contain a valid string representation of a date and *expression2* a valid string representation of time. For example:

VALUES(TIMESTAMP('2003-04-15', '08:20:15'))

2003-04-15-08.20.15.000000

TIMESTAMP_FORMAT(string,format)

Returns a timestamp value after extracting it from *string*. The second argument, *format*, contains the timestamp format used in *string*. (For details on the timestamp format, see the *VARCHAR_FORMAT* function.) For example:

VALUES(TIMESTAMP_FORMAT('2003-4-15 08:20:15', 'YYYY-MM-DD HH24:MI:SS'))

2003-04-15-08.20.15.000000

TIMESTAMP_ISO(expression)

Returns a timestamp value from *expression*. If *expression* is a date value, then the time value will consist of zeros. If the *expression* is a time value, then the current date will be used in constructing the timestamp value. For example:

VALUES(TIMESTAMP_ISO('2003-04-15')) -> 2003-04-15-00.00.00.000000

TIMESTAMPDIFF(interval_type, tsdiff_expression)

Returns an estimated number of intervals between two timestamp values. The *interval_type* argument must be a power of 2 in the range between 1 and 256, where the meaning of the value is interpreted as in the following table:

interval_type	Timestamp difference returned
1	Fractions of a second
2	Seconds
4	Minutes
8	Hours

16	Days
32	Weeks
64	Months
128	Quarters
256	Years

The *tsdiff_expression* must be the result of subtracting two timestamp values and converting to a single *CHAR(22)* value. For example:

VALUES(TIMESTAMPDIFF(2, CHAR(CURRENT_DATE - CURRENT_TIMESTAMP)))

515

TO_CHAR(timestamp, format)

Synonym for *VARCHAR_FORMAT*.

TO_DATE(string, format)

Synonym for *TIMESTAMP_FORMAT*.

TRANSLATE(string, tostring, fromstring [,pad_char])

Returns a string equivalent to the *string* argument with each character found in *tostring* replaced with the corresponding character from *fromstring*. The optional *pad_char* argument can contain a single-byte character that will be used to pad *tostring* if it is shorter than *fromstring*. If the same character appears more than once in *fromstring*, then only the first occurrence will count and all following occurrences will be ignored. For example:

VALUES(TRANSLATE('123,456,789.45', ',', '.')) -> '123.456.789,45'

TRUNCATE(number1, number2) or *TRUNC(number1, number2)*

Returns *number1* with all digits to the right of the decimal point by *number2* places replaced with zeros. If *number2* is negative, then the truncation occurs to the left of the decimal place. For example:

VALUES(TRUNCATE(DEC('123.456'), -1)) -> 120.

TYPE_ID(expression)

Returns the internal type identifier to the user-defined structured type instance stored in *expression*. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

TYPE_NAME(expression)

Returns the unqualified name to the user-defined structured type instance stored in *expression*. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

TYPE_SCHEMA(expression)

Returns the schema name to the user-defined structured type instance stored in *expression*. For more information on User Defined Types (UDTs) please consult the DB2 User Guide.

VALUE(expression [,...])

Returns the first argument that is not NULL. For example:

VALUES(VALUE('Hello!', 5)) -> 'Hello!'

VAR or *VARIANCE*({ *ALL* | *DISTINCT* } *expression*)

Returns the variance of the values contained within *expression*. The *ALL* keyword specifies the default behavior in which all values in *expression* are used in the calculation. The keyword *DISTINCT* omits duplicate values in *expression* when doing the calculation. For example:

SELECT VARIANCE(values) FROM NUMBERS -> 987244882.22

VARCHAR(*expression* [, *length*])

Returns a varying-length character representation of *expression*, where *length* is the length attribute for the varying-length character result. If *length* is omitted, then the length of the result is the length of the expression.

VARCHAR_FORMAT(*expression*, *format*)

Returns a string representation of the timestamp value in *expression* in the *format* provided in the *format* argument. Listed below are the available specifiers for *format* and their meanings.

YYYY

Four-digit year

MM

Two-digit month (01-12)

DD

Two-digit day of month (01, 02, ...)

HH24

Two-digit hour of day (00-24)

MI

Two-digit minute (00-59)

SS

Two-digit second (00-59)

For example:

VALUES(VARCHAR_FORMAT(CURRENT_TIMESTAMP, 'YYYY-MM-DD HH24:MI:SS'))

2003-06-24 09:37:45

VARGRAPHIC(*expression* [, *length*])

Returns a varying-length graphic string representation of *expression*, where *length* is the length attribute for the varying-length graphic string result. If *length* is omitted, then the length of the result is the length of the expression.

WEEK(*expression*)

Returns the week of the year for the date value contained in *expression*. The range for the result is 1-54 and the week starts with Sunday. For example:

VALUES(WEEK('2003-04-15')) -> 16

WEEK_ISO(expression)

Returns the week of the year for the date value contained in *expression*. The range for the result is 1-53, and the week starts with Monday. The first week of the year always contains January 4th. For example:

VALUES(WEEK_ISO('2003-04-15')) -> 16

YEAR(expression)

Returns the year part of the date value in *expression*. For example:

VALUES(YEAR('2004-04-15')) -> 2004

4.5.2 MySQL-Supported Functions

This section provides an alphabetical listing of MySQL-supported functions, with examples and corresponding results.

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

SELECT ACOS(0) -> 1.570796

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

SELECT ASCII('x') -> 120

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

SELECT ASIN(0) -> 0.000000

ATAN(number)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

SELECT ATAN(3.1415) -> 1.262619

ATAN2(x,y)

Returns the arctangent of the two variables *x* and *y*. *ATAN2(x,y)* is similar to *ATAN(y/x)*, with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

ATAN2(3.1415, 1) -> 1.262619

BENCHMARK(*count*,*expr*)

Executes the expression (*expr*) *count* times. The result value is always 0. For example:

BENCHMARK(1000000,ATAN2(3.1415, 1)) -> 0

BIN(*number*)

Returns a string containing the binary value of *n*, where *n* is a *BIGINT* number.

BINARY *string*

Casts *string* to a binary string.

BIT_AND(*expr*)

An aggregate function that returns the bitwise *AND* of all bits in *expr*. The calculation is performed with 64-bit (*BIGINT*) precision. The value of -1 is returned when no matching rows are found. For example:

BIT_AND(mycolumn) -> 0

BIT_COUNT(*number*)

Returns the number of bits that are set in *number*. For example:

BIT_COUNT(5) -> 2

BIT_OR(*expr*)

An aggregate function that returns the bitwise *OR* of all bits in *expr*. The calculation is performed with 64-bit (*BIGINT*) precision. The value of zero is returned when no matching rows are found. For example:

BIT_OR(mycolumn) -> 1

CHAR(*number* [,...])

Returns a string consisting of the characters given by the ASCII code values in the arguments. Any NULL values are ignored. For example:

CHAR(120,121,122) -> 'xyz'

COALESCE(*list*)

Returns the first non-NULL element in the list. For example:

COALESCE(NULL, 1, 2) -> 1

COMPRESS(*string*)

Returns a compressed version of *string*.

CONCAT_WS(*separator*, *str1*, *str2*[,...])

A special form of **CONCAT**() that inserts *separator* between every pair of string arguments concatenated. If *separator* is NULL, then the result is NULL. For example:

CONCAT_WS(' ', au_lname, au_fname) -> 'Jefferson, Thomas'

CONNECTION_ID()

Returns the connection ID for the connection. Every connection has its own unique ID. For example:

CONNECTION_ID() -> 305102

CONV(*number,from_base,to_base*)

Returns a string representation of the number *number*, converted from base *from_base* to base *to_base*. If any argument is NULL, then the result is NULL. For example:

CONV(12,10,2) -> 1100

COS(*number*)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

CURDATE()

Returns today's date as a value in YYYY-MM-DD or YYYYMMDD format, depending on whether the function is used in a string or numeric context. For example:

CURDATE() -> '2003-06-24'

CURTIME()

Returns the current time as a value in HH:MM:SS or HHMMSS format, depending on whether the function is used in a string or numeric context. For example:

CURTIME() -> '20:40:20'

DATABASE()

Returns the current database name. For example:

DATABASE() -> 'PUBS'

DATE_ADD(*date,INTERVAL expr type*)

DATE_SUB(*date,INTERVAL expr type*)

ADDDATE(*date,INTERVAL expr type*)

SUBDATE(*date,INTERVAL expr type*)

These functions perform date arithmetic calculations. *ADDDATE()* and *SUBDATE()* are synonyms for *DATE_ADD()* and *DATE_SUB()*. *DATE_ADD()* returns the result of adding the *INTERVAL* to the *date* expression. *DATE_SUB()* is the result of subtracting the *INTERVAL* from the *date* expression. For example:

DATE_ADD('1999-04-15', INTERVAL 1 DAY) -> '1999-04-16'

DATE_SUB('1999-04-15', INTERVAL 1 DAY) -> '1999-04-14'

DATE_FORMAT(date, format)

Formats the *date* value according to the *format* string. Listed below are the available specifiers for *format* and their meanings.

%a

Abbreviation of the day (Sun-Sat)

%b

Abbreviation of the month (Jan-Dec)

%c

Month number (1-12)

%D

Day of month with a suffix (1st, 2nd, 3rd, ...)

%d

Two digit day of month (01, 02, ...)

%e

Day of month (1, 2, 3, ...)

%H

Hour (00-23)

%h

Hour (01-12)

%i

Minutes (00-59)

%I

Hour (01-12)

%j

Day of year (001-366)

%k

Hour (0-23)

%l

Hour (1-12)

%M

Full month name (January-December)

%m

Month (01-12)

%p

A.M. or P.M.

%r

12-hour time (hh:mm:ss A.M or P.M)

%S, %s

Seconds (00-59)

%T

24-hour time (hh:mm:ss)

%U

Week number (00-53, Sunday being the first day of the week)

%u

Week number (00-53, Monday being the first day of the week)

%V

Week number (01-53, Sunday being the first day of the week)

%v

Week number (01-53, Monday being the first day of the week)

%W

Name of the day (Sunday-Saturday)

%w

Day of the week (0 = Sunday, 6 = Saturday)

%X

Four-digit year with Sunday being the first day of the week

`%x`

Four-digit year with Monday being the first day of the week

`%Y`

Four-digit year

`%y`

Two-digit year

`%%`

Literal "%"

For example:

`DATE_FORMAT('1999-04-15', '%M-%D-%Y')` -> 'April-15th-1999'

DAYNAME(date)

Returns the name of the weekday for *date*. For example:

`DAYNAME('1999-04-15')` -> 'Thursday'

DAYOFMONTH(date)

Returns the day of the month for *date*, in the range 1 to 31. For example:

`DAYOFMONTH('1999-04-15')` -> 15

DAYOFWEEK(date)

Returns the weekday index for *date* (1 = Sunday, 2 = Monday, . . . 7 = Saturday). For example:

`DAYOFWEEK('1999-04-15')` -> 5

DAYOFYEAR(date)

Returns the day of the year for *date*, in the range 1 to 366. For example:

`DAYOFYEAR('1999-04-15')` -> 105

DECODE(crypt_str, pass_str)

Decrypts the encrypted string *crypt_str* using *pass_str* as the password; *crypt_str* should be a string returned from *ENCODE()*. For example:

`DECODE(ENCODE('foo','bar'),'bar')` -> 'foo'

DEGREES(number)

Returns the argument *number* converted from radians to degrees. For example:

`DEGREES(3.1415926)` -> 179.99999692953

FORMAT(12345.2132,2) -> 12,345.21

FORMAT(12345.2132,0) -> 12,345

FROM_DAYS(number)

Given a day number *number*, returns a *DATE* value. This function should not be used for values that precede the advent of the Gregorian calendar (1582), due to the days lost when the calendar was changed. For example:

FROM_DAYS(888888) -> 2433-09-10

FROM_UNIXTIME(unix_timestamp)

Returns a representation of the *unix_timestamp* argument as a value in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. For example:

FROM_UNIXTIME(888123892) -> 1998-02-21 21:04:52

FROM_UNIXTIME(unix_timestamp, format)

Returns a string representation of the *unix_timestamp*, formatted according to the *format* string. *format* may contain the same specifiers as those listed in the entry for the *DATE_FORMAT()* function. For example:

FROM_UNIXTIME(888123892, '%Y %D %M') -> '1998 21st February'

GET_LOCK(str, timeout)

Tries to obtain a lock with a name given by the string *str*, with a time-out of *timeout* seconds. Returns 1 if the lock is obtained successfully or NULL if an error occurs or the attempt to acquire the lock times out. For example:

GET_LOCK('lochness',10) -> 1

GREATEST(x,y[,...])

Returns the largest argument. For example:

GREATEST(8,2,4) -> 8

HEX(number)

Returns a string representation of the hexadecimal value of *number*. This is equivalent to *CONV(number,10,16)*. For example:

HEX(255) -> FF

HOUR(time)

Returns the hour for *time*, in the range 0 to 23. For example:

HOUR('08:20:15') -> 8

IF(expr1, expr2, expr3)

Returns *expr2* if *expr1* is TRUE, otherwise returns *expr3*. For example:

IF(1,'yes','no') -> 'yes'

IF(0,'yes','no') -> 'no'

IFNULL(expr1, expr2)

Returns *expr1* if *expr1* is not NULL, otherwise returns *expr2*. For example:

IFNULL(0,'NULL') -> 0

IFNULL(NULL,'NULL') -> 'NULL'

INSERT(str, pos, len, newstr)

Returns the string *str* with *newstr* inserted at character position *pos* for length *len*. For example:

INSERT('paper',2,3,'ea') -> 'pear'

INSTR(str,substr)

Returns the position of the first occurrence of the substring *substr* in the string *str*. For example:

INSTR('ducks','c') -> 3

INTERVAL(num1,num2,num3,num4[,...n])

Returns 0 if *num1* < *num2*, 1 if *num1* < *num3*, and so on. It is required that *num2* < *num3* < *num4* < . . . < *numN*. For example:

INTERVAL(5,1,6) -> 1

INTERVAL(5,2,3,7,9) -> 2

IS_FREE_LOCK(lock)

Returns 1 if *lock* is free and zero if the lock is currently in use. The function may return NULL on error conditions. For example:

IS_FREE_LOCK('lochness') -> 0

ISNULL(expr)

If *expr* is NULL, *ISNULL()* returns 1; otherwise it returns 0. For example:

ISNULL(1) -> 0

ISNULL(NULL) -> 1

LAST_INSERT_ID([expr])

Returns the last automatically generated value that was inserted into an *AUTO_INCREMENT* column. For example:

LAST_INSERT_ID() -> 0

LCASE(str)

Synonym for *lower(str)*. For example:

LCASE('DUCK') -> 'duck'

LEAST(X,Y[,...n])

With two or more arguments, returns the smallest (minimum-valued) argument. For example:

LEAST(10,5,3,7) -> 3

LEFT(str,len)

Returns the leftmost *len* characters from the string *str*. For example:

LEFT('Ducks', 4) -> 'Duck'

LENGTH(str)

These functions return the length of the string *str*. For example:

LENGTH('DUCK') -> 4

LOAD_FILE(file_name)

Reads the file and returns the file contents as a string. The file must be on the server, and the user must specify the full pathname to the file and have access to the file.

LOCATE(substr,str), POSITION(substr IN str)

Returns the position of the first occurrence of the substring *substr* in the string *str*. Returns zero if *substr* is not in *str*. *LOCATE* is a synonym for the standard *POSITION(substr IN str)*. For example:

LOCATE('al','Donald') -> 4

POSITION('al' IN 'Donald') -> 4

LOCATE(substr,str,pos)

Returns the position of the first occurrence of the substring *substr* in the string *str*, starting at position *pos*; returns zero if *substr* is not in *str*. For example:

LOCATE('World', 'Hello, World!') -> 8

LOG(X)

Returns the natural logarithm of *X*. For example:

LOG(50) -> 3.912023

LOG2(X)

Returns the base-2 logarithm of *X*. For example:

LOG2(50) -> 5.64386

LOG10(X)

Returns the base-10 logarithm of *X*. For example:

LOG10(50) -> 1.698970

LPAD(str,len,padstr)

Returns the string *str*, left-padded with the string *padstr* until *str* is *len* characters long. For example:

LPAD('ucks',6,'d') -> 'dducks'

LTRIM(str)

Returns the string *str* with leading-space characters removed. For example:

LTRIM(' Howdy! ') -> 'Howdy! '

MAKE_SET(bits, str1, str2[, ...n])

Returns a *set* (a string containing substrings separated by commas) consisting of the string arguments that have the corresponding bit in *bits* set; *str1* corresponds to bit 0, *str2* to bit 1, etc. NULL strings in *str1*, *str2*, ... are not appended to the result. For example:

MAKE_SET(1 | 4,'hello','nice','world') -> 'hello,world'

MD5(string)

Calculates an MD5 checksum for the *string*. Value is returned as a 32-digit-long hex number. For example:

MD5('somestring') -> 1f129c42de5e4f043cbd88ff6360486f

MINUTE(time)

Returns the minute for *time*, in the range to 59. For example:

MINUTE('08:20:15') -> 20

MONTH(date)

Returns the month for *date*, in the range 1 to 12. For example:

MONTH('1999-04-15') -> 4

MONTHNAME(date)

Returns the name of the month for *date*. For example:

MONTHNAME('1999-04-15') -> 'April'

NOW() , SYSDATE()

Returns the current date and time as a value in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. For example:

NOW() -> 2003-06-24 20:40:24

SYSDATE() -> 2003-06-24 20:40:24

CURRENT_TIMESTAMP -> 2003-06-24 20:40:24

NULLIF(expr1, expr2)

Returns NULL if *expr1* is equal to *expr2*; otherwise returns *expr1*. For example:

NULLIF(2,29) -> 2

NULLIF(29,29) -> NULL

OCT(n)

Returns an octal value equivalent of *n*, where *n* is a number. This is equivalent to *CONV(N,10,8)*. Returns NULL if *n* is NULL. For example:

OCT(255) -> 377

ORD(str)

Returns the character ordinal of the multibyte character string *str*. The value is calculated using the following formula: ((first byte ASCII code)*256+(second byte ASCII code)*256*256)(third byte ASCII code)*256*256*256[,...]. If *str* isn't a multibyte character, then this function returns the same value as the *ASCII()* function. For example:

ORD('29') -> 50

PASSWORD(str)

Calculates a password string from the plain-text password *str*. This is the function that is used for encrypting MySQL passwords. For example:

PASSWORD('password') -> 5d2e19393cc5ef67

PERIOD_ADD(period, months)

Adds the number of months found in *months* to the period in *period* (in the format YMMM or YYYYMM). Returns a value in the format YYYYMM. For example:

PERIOD_ADD(9902,3) -> 199905

PERIOD_DIFF(period1, period2)

Returns the number of months between *period1* and *period2*. *period1* and *period2* should be in the format YMMM or YYYYMM. For example:

PERIOD_DIFF(9902,9905) -> -3

PI()

Returns the value of π . For example:

PI() -> 3.141593

POW(X, Y), POWER(X, Y)

Returns the value of *X* raised to the power of *Y*. For example:

POW(2, 8) -> 256.000000

QUARTER(date)

Returns the quarter of the year for *date*, in the range 1 to 4. For example:

QUARTER('1999-04-15') -> 2

RADIANS(X)

Returns the argument *X*, converted from degrees to radians. For example:

RADIANS(180) -> 3.1415926535898

RAND() , RAND(N)

Returns a random floating-point value in the range 0 to 1.0. If an integer argument *N* is specified, it is used as the seed value. For example:

RAND() -> 0.29588872501244

RELEASE_LOCK(str)

Releases the lock named by the string *str* that was obtained with *GET_LOCK()*. Returns 1 if the lock is released or NULL if the named lock doesn't exist or isn't locked by this thread (in which case the lock is not released). For example:

RELEASE_LOCK('lochness') -> 1

REPEAT(str, count)

Returns a string consisting of the string *str* repeated *count* times. For example:

REPEAT('Duck', 3) -> 'DuckDuckDuck'

REPLACE(str, from_str, to_str)

Returns the string *str* with all occurrences of the string *from_str* replaced by the string *to_str*. For example:

REPLACE('change', 'e', 'ing') -> 'changing'

REVERSE(str)

Returns the string *str* reversed. For example:

REVERSE('STOP') -> 'POTS'

RIGHT(str, ten)

Returns the rightmost 10 characters from the string *str*. For example:

RIGHT('Hello, World!', 6) -> 'World!'

ROUND(X[,D])

Returns the argument *X*, rounded to a number with *D* decimals. If *D* is 0, the result has no decimal point or fractional part. For example:

ROUND(12345.6789, 2) -> 12345.68

RPAD(str,len,padstr)

Returns the string *str*, right-padded with the string *padstr* until *str* is *len* characters long. For example:

RPAD('duck',6,'s') -> 'duckss'

RTRIM(str)

Returns the string *str* with trailing space characters removed. For example:

RTRIM(' welcome ') -> 'welcome '

SEC_TO_TIME(seconds)

Returns the *seconds* argument, converted to hours, minutes, and seconds, as a value in HH:MM:SS or HHMMSS format, depending on whether the function is used in a string or numeric context. For example:

SEC_TO_TIME(256) -> 00:04:16

SECOND(time)

Returns the second for *time*, in the range to 59. For example:

SECOND('08:20:15') -> 15

SHA(X) or *SHA1(X)*

Returns a SHA1 160-bit checksum for *X*. For example:

SHA('abc') -> 'a9993e364706816aba3e25717850c26c9cd0d89d'

SIGN(X)

Returns the sign of the argument as -1, 0, or 1, depending on whether *X* is negative, zero, or positive. For example:

SIGN(-3.1415926) -> -1

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SOUNDEX(str)

Returns a soundex string from *str*. For example:

SOUNDEX('thimble') -> 'T514'

expr1 SOUNDS LIKE *expr2*

Synonymous with the expression:

SOUNDEX(*expr1*) = SOUNDEX(*expr2*)

SPACE(n)

Returns a string consisting of *n* space characters. For example:

SPACE(5) -> ' '

STD(expr), STDDEV(expr)

Returns the standard deviation of *expr*. The *STDDEV()* form of this function is provided for Oracle compatibility. For example:

STD(5) -> NULL

STRCMP(expr1, expr2)

STRCMP() returns zero, as marked if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise. For example:

STRCMP('DUCKY', 'DUCK') -> 1

STRCMP('DUCK', 'DUCK') -> 0

SUBSTRING(str, pos, len), MID(str, pos, len)

Returns a substring len characters long from string *str*, starting at position *pos*. These functions are synonyms for the ANSI SQL92 function *SUBSTRING(str FROM pos FOR len)*. For example:

SUBSTRING('Hello, World!', 8, 10) -> 'World!'

SUBSTRING('Hello, World!' FROM 8 FOR 10) -> 'World!'

SUBSTRING(str, pos), SUBSTRING(str FROM pos)

Returns a substring from string *str* starting at position *pos*. For example:

SUBSTRING('Hello, World!', 8) -> 'World!'

SUBSTRING('Hello, World!' FROM 8) -> 'World!'

SUBSTRING_INDEX(str, delim, count)

Returns the substring of *str* after *count* occurrences of the delimiter *delim*. For example:

SUBSTRING_INDEX('www.mysql.com', '.', 2) -> 'www.mysql'

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

SELECT TAN(3.1415) -> -0.000093

TIME_FORMAT(time, format)

This is used like *DATE_FORMAT()*, but the *format* string may contain only those format specifiers that handle hours, minutes, and seconds. Other specifiers produce a NULL value or 0. See *DATE_FORMAT* for the format specifiers available. For example:

TIME_FORMAT('2003-04-15 08:20:15', '%r') -> 08:20:15 AM

TIME_TO_SEC(time)

Returns the *time* argument, converted to seconds. For example:

TIME_TO_SEC('08:20:15') -> 30015

TO_DAYS(date)

Given a *date*, returns a day number (the number of days since the year 0). For example:

TO_DAYS('1999-04-15') -> 730224

TRUNCATE(X,D)

Returns the number *X*, truncated to *D* decimals. If *D* is 0, the result has no decimal point or fractional part. For example:

TRUNCATE('123.456', 2) -> '123.45'

TRUNCATE('123.456', 0) -> '123'

TRUNCATE('123.456', -1) -> '120'

UCASE(str)

Synonym for *UPPER(str)*. For example:

UCASE('duck') -> 'DUCK'

UNCOMPRESS(string)

Returns a compressed version of *string*.

UNHEX(str)

Returns a binary string constructed from hex characters in *str*.

UNIX_TIMESTAMP() , *UNIX_TIMESTAMP(date)*

If called with no argument, returns a Unix timestamp (seconds since 1970-01-01 00:00:00 GMT). If *UNIX_TIMESTAMP()* is called with a *date* argument, it returns the value of the argument as seconds since 1970-01-01 00:00:00 GMT. For example:

UNIX_TIMESTAMP() -> 1056512427

UNIX_TIMESTAMP('1999-04-15') -> 924159600

USER() , *SYSTEM_USER()* , *SESSION_USER()*

These functions return the current MySQL username. For example:

USER() -> 'login@machine.com'

SYSTEM_USER() -> 'login@machine.com'

SESSION_USER() -> 'login@machine.com'

VERSION()

Returns a string indicating the MySQL server version. For example:

VERSION() -> '4.0.12-standard'

WEEK(date), WEEK(date, first)

With a single argument, returns the week for *date*, in the range 1 to 53. (The beginning of a week 53 is possible during some years.) The two-argument form of *WEEK()* allows the user to specify whether the week starts on Sunday (0) or Monday (1). For example:

WEEK('1999-04-15') -> 15

WEEKDAY(date)

Returns the weekday index for *date* (0 = Monday, 1 = Tuesday, . . . 6 = Sunday). For example:

WEEKDAY('1999-04-15') -> 3

YEAR(date)

Returns the year for *date*, in the range 1000 to 9999. For example:

YEAR('1999-04-15') -> 1999

YEARWEEK(date), YEARWEEK(date, first)

Returns the year and week for *date*. The second argument works exactly like the second argument to *WEEK()*. Note that the year may be different from the year in the *date* argument for the first and the last week of the year. For example:

YEARWEEK('1999-04-15') -> 199915

4.5.3 Oracle-Supported Functions

This section provides an alphabetical listing of the SQL functions specific to Oracle, with examples and corresponding results.

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to 1 and is expressed in radians. For example:

SELECT ACOS(0) FROM DUAL -> 1.570796

ADD_MONTHS(date, int)

Returns the *date* plus *int* months. For example:

SELECT ADD_MONTHS('15-APR-1999', 3) FROM DUAL -> 15-JUL-99

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

SELECT ASCII('x') FROM DUAL -> 120

ASCIISTR(*text*)

Converts *text* from any character set into an ASCII equivalent. Characters in *text* that have no equivalent in ASCII will be replaced with the string \XXXX, where XXXX represents the UTF-16 code unit. For example:

```
SELECT ASCIISTR('ÄBC') FROM DUAL -> '\00C4BC'
```

ASIN(*number*)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from -/2 to /2 and is expressed in radians. For example:

```
SELECT ASIN( 0 ) FROM DUAL -> 0.000000
```

ATAN(*number*)

Returns the arctangent of *number*. The resulting value ranges from -/2 to /2 and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) FROM DUAL -> 1.262619
```

ATAN2(*number, nbr*)

Returns the arctangent of *number* and *nbr*. The values for *number* and *nbr* are not restricted, but the results range from - to and are expressed in radians. ATAN2(*x,y*) is similar to ATAN(*y/x*), with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
SELECT ATAN2(3.1415, 1) FROM DUAL -> 1.26261873
```

BFILENAME(*directory, filename*)

Returns a *BFILE* locator associated with a physical *LOB* binary file on the server's filesystem in *directory* with the name *filename*.

BIN_TO_NUM(*expr[,...n]*)

Returns a decimal number equivalent of the binary bit vector contained in the *expr* arguments. For example:

```
SELECT BIN_TO_NUM(1,0,1) FROM DUAL -> 5
```

BITAND(*integer1, integer2*)

Returns the bitwise *AND* of the two integer arguments. For example:

```
SELECT BITAND(101, 2) FROM DUAL -> 0
```

```
SELECT BITAND(column1, 1) FROM DUAL -> 1
```

CARDINALITY(*nested_table*)

Returns the number of elements (cardinality) of the *nested_table*. If the *nested_table* is empty, NULL will be returned. For example:

```
SELECT CARDINALITY(mytable) FROM DUAL -> 6
```

CHARTOROWID(*char*)

Converts a value from a character datatype (*CHAR* or *VARCHAR2* datatype) to a *ROWID* datatype.

CHR(number [USING NCHAR_CS])

Returns the character having the binary equivalent to *number* in either the database character set (if *USING NCHAR_CS* is not included) or the national character set (if *USING NCHAR_CS* is included).

COALESCE(list)

Returns the first non-NULL element in the list. For example:

SELECT COALESCE(NULL, 1, 2) FROM DUAL -> 1

COLLECT (column)

Creates for each group a nested table consisting of all values in a column. This is an aggregate function.

COMPOSE(string)

Returns *string* as a fully normalized UNICODE string.

CONCAT(string1, string2)

Returns *string1* concatenated with *string2*. *CONCAT* is equivalent to the concatenation operator (||). For example:

SELECT CONCAT(au_lname, au_fname) FROM AUTHORS -> 'JeffersonThomas'

CONVERT(char_value, target_char_set, source_char_set)

Converts a character string from one character set to another; returns *char_value* in the *target_char_set* after converting *char_value* from the *source_char_set*.

CORR_K(expr1, expr2 [,return_type])

CORR_S(expr1, expr2 [,return_type])

CORR_K returns Kendall's tau-b correlation coefficient, and *CORR_S* returns Spearman's rho correlation coefficient of a set of numbered pairs (*expr1* and *expr2*). The *return_type* argument, a *VARCHAR2*, can be omitted or one of the following values: 'COEFFICIENT', 'ONE_SIDED_SIG', 'TWO_SIDED_SIG'. The value of 'COEFFICIENT', which specifies the default behavior, returns the coefficient of the correlation. The values of 'ONE_SIDED_SIG' and 'TWO_SIDED_SIG' will return the one- and two-tailed significance of the correlation, respectively.

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) FROM DUAL -> 1.000000

COSH(number)

Returns the hyperbolic cosine of number. For example:

SELECT COSH(180) FROM DUAL -> 7.4469E+77

SELECT COT(3.1415) FROM DUAL -> -10792.88993953

COT(number)

Returns the cotangent of *number*. For example:

```
SELECT COT( 3.1415 ) -> -10792.88993953
```

CV([*dimension_column*])

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns the current value of the *dimension_column*. *CV* can only be used in the righthand side of a rule, since it returns the value of the *dimension_column* from the left-hand side of the same rule.

DBTIMEZONE

Returns the time zone offset from UTC time for the database server. For example:

```
SELECT DBTIMEZONE FROM DUAL -> +00:00
```

DECODE(*expr*, *search*, *result* [, *search*, *result* [,...*n*]] [, *default*])

Compares *expr* to the *search* value; if *expr* is equal to a *search*, it returns the result. For example:

```
DECODE ('B','A',1,'B',2,...'Z',26,'?') -> 2
```

Without a match, *DECODE* returns *default*, or NULL if *default* is omitted. Refer to Oracle documentation for more details. Consider using *CASE* instead, as *CASE* is part of the ANSI/ISO SQL standard.

DECOMPOSE(*string* [{CANONICAL | COMPATIBILITY}])

Returns *string* decomposed into UNICODE code-points. The second argument specifies the type of decomposition performed. *CANONICAL*, which specifies the default behavior, allows the original UNICODE string to be recomposed.

DEPTH(*number*)

Returns the depth of the path specified by the *UNDER_PATH* condition in an XML query. See the Oracle SQL Reference for more information.

DEREF(*expression*)

Returns the object referenced by *expression*, where *expression* must return a *REF* to an object.

DUMP(*expression* [, *return_format* [, *starting_at* [, *length*]]])

Returns a *VARCHAR2* value containing a datatype code, *length* in bytes, and internal representation of *expression*. The resulting value is returned in the format of *return_format*. For example:

```
SELECT DUMP('abc', 1016) FROM DUAL
```

```
Typ=96 Len=3 CharacterSet=AL32UTF8: 61,62,63
```

EMPTY_BLOB() , *EMPTY_CLOB*()

Returns an empty *LOB* locator that can be used to initialize a *LOB* variable. It can also be used to initialize a *LOB* column or attribute to empty in an *INSERT* or *UPDATE* statement.

EXISTSNODE(*instance*, *xpath* [, *namespace*])

Returns 1 if applying the XPath query in *xpath* would return any nodes from *instance*; otherwise, returns 0. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference.

EXTRACT(*instance*, *xpath* [, *namespace*])

Returns the XML nodes from *instance* returned by running the XPath query contained in the *xpath* parameter. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

```
SELECT EXTRACT( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),  
                '/foo/bar' ) from DUAL
```

```
<bar>Hello, World!</bar>
```

EXTRACTVALUE(*instance*, *xpath* [, *namespace*])

Returns the value from an XML node returned by running the XPath query contained in the *xpath* parameter. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

```
SELECT EXTRACTVALUE( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),  
                    '/foo/bar' ) from DUAL
```

```
Hello, World!
```

FIRST

An aggregate function that returns a specified value from the row that ranks first, given the order specified in the *ORDER BY* clause. Syntax:

```
aggregate(aexpr) KEEP (DENSE_RANK FIRST ORDER BY expr [,...n])
```

where the syntax of *expr* is:

```
expr := [ASC|DESC] [NULLS {FIRST|LAST}]
```

The first ranking row following the order specified by *expr* will be used in the aggregate function *aggregate*. The *aexpr* is the expression passed to the aggregate function. For example:

```
SELECT MAX(c1) KEEP (DENSE_RANK FIRST ORDER BY c2) FROM FIVE_NUMS
```

```
1
```

FIRST_VALUE(*expression* IGNORE NULLS) *OVER* (*window_clause*)

Returns the first value in an ordered set of values. *FIRST_VALUE* is an analytic function. See the [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

```
SELECT FIRST_VALUE(col1) OVER ( ) FROM NUMS
```

```
1
```

```
1
```

```
1
```

```
1
```

FROM_TZ(*timestamp*, *timezone*)

Returns *timestamp* converted to a *TIMESTAMP WITH TIME ZONE* value, where *timestamp* is a *TIMESTAMP* value and *timezone* is a string in the TZH:TZM format. For example:

```
SELECT FROM_TZ(TIMESTAMP '2004-04-15 23:59:59', '8:00') FROM DUAL
```

'15-APR-04 11.59.59 PM +08:00'

GREATEST(expression [,...n])

Returns the greatest of the list of *expressions*. All *expressions* after the first are implicitly converted to the datatype of the first expression before the comparison. For example:

SELECT GREATEST(8,2,4) FROM DUAL -> 8

GROUP_ID()

Returns a positive value for each duplicate group returned by a query containing a *GROUP BY* clause. This function is useful in filtering out duplicate groups created when using *CUBE*, *ROLLUP*, or other *GROUP BY* extension (see *GROUPING*.)

GROUPING(column_name)

Returns 1 when a row is added by *CUBE*, *ROLLUP*, or other *GROUP BY* extensions; otherwise returns 0. For example:

**SELECT royalty, SUM(advance) 'total advance',
GROUPING(royalty) 'grp'
FROM titles
GROUP BY royalty WITH ROLLUP**

royalty	total advance	grp
-----	-----	---
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

GROUPING_ID(column_name1 [, column_name2 ,...])

Returns the base-10 number that is equal to the binary value constructed by concatenating the *GROUPING* values on each of the parameters. *GROUPING_ID* is useful when returning a query containing multiple levels of aggregation created by *GROUP BY* expressions. Consider using the *GROUPING_ID* function instead of multiple *GROUPING* functions within one query. This function is a shorthand equivalent of:

BIN_TO_NUM(GROUPING(column_name1) [, GROUPING(column_name2) ,...])

HEXTORAW(string)

Converts *string* containing hexadecimal digits into a raw value. For example:

SELECT HEXTORAW('0FE') FROM DUAL -> '00FE'

INITCAP(string)

Returns *string*, with the first letter of each word in uppercase and all other letters in lowercase. For example:

```
SELECT INITCAP('thomas jefferson') FROM DUAL -> 'Thomas Jefferson'
```

INSTR(*string1*, *string2* [, *start_at* [, *occurrence*]])

Returns the position of *string2* within *string1*. *INSTR* searches *string1* with a starting position of *start_at* (an integer) looking for the specified *occurrence* of *string2*. For example:

```
SELECT INSTR('foobar', 'o', 1, 1) FROM DUAL -> 2
```

Use *INSTRB* for bytes, *INSTRC* for UNICODE complete characters, *INSTR2* for UNICODE UCS2 code points, and *INSTR4* for UNICODE UCS4 code points.

ITERATION_NUMBER

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns the number of times the rules within the *MODEL* clause have been executed while processing the query.

LAG(*expression* [, *offset*] [, *default*]) *OVER* (*window_clause*)

An analytic function that provides access to more than one row of a table at the same time without a self join. *LAG* provides a "lagging" value in the result set that lags *offset* rows behind the current row. The *default* value is used for the first *offset* rows in the result set, since the "lagging" value is undefined for these rows. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

```
SELECT c1, LAG(c1, 2, 0) OVER (ORDER BY c1) FROM FIVE_NUMS
```

```
1    0
2    0
3    1
4    2
5    3
```

LAST

Returns the row that ranks last given the order specified in the *ORDER BY* clause. The syntax is:

```
aggregate(aexpr) KEEP (DENSE_RANK LAST ORDER BY expr [,...n])
```

where the syntax of *expr* is:

```
expr := [ASC|DESC] [NULLS {FIRST|LAST}]
```

The last ranking row following the order specified by *expr* will be used in the aggregate function *aggregate*. The *aexpr* is the expression passed to the aggregate function. For example:

```
SELECT MIN(c1) KEEP (DENSE_RANK LAST ORDER BY c1) FROM FIVE_NUMS
```

```
5
```

LAST_DAY(*date*)

Returns the date of the last day of the month that contains *date*. For example:

```
SELECT LAST_DAY('15-APR-1999') FROM DUAL -> 30-APR-99
```

LAST_VALUE(expression [IGNORE NULLS]) OVER (window_clause)

Returns the last value in an ordered set of values. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

SELECT c1, LAST_VALUE(c1) OVER (ORDER BY c1) FROM FIVE_NUMS

```
1    5
2    5
3    5
4    5
5    5
```

LEAD(expression [,offset][,default]) OVER(window_clause)

An analytic function that provides access to more than one row of a table at the same time without a self join. *LEAD* provides a "leading" value in the result set that is *offset* rows ahead of the current row. The *default* value is used for the last *offset* rows in the result set, since the "leading" value is undefined for these rows. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *window_clause*. For example:

SELECT c1, LEAD(c1, 2) OVER (ORDER BY c1) FROM FIVE_NUMS

```
1    3
2    4
3    5
4
5
```

LEAST(expression [,...n])

Returns the least of the list of expressions. For example:

SELECT LEAST(10,5,3,7) FROM DUAL -> 3

LENGTH(string)

Returns the integer length of *string*, or NULL if *string* is NULL. For example:

SELECT LENGTH('DUCK') FROM DUAL -> 4

LENGTHB(string)

Returns the length of *char* in bytes; otherwise, the same as *LENGTH*. For example:

SELECT LENGTHB('DUCK') FROM DUAL -> 4

Use *LENGTHB* for bytes, *LENGTHC* for UNICODE complete characters, *LENGTH2* for UNICODE UCS2 code points, and *LENGTH4* for UNICODE UCS4 code points.

LNNVL(condition)

Returns *true* if *condition* is false or if one of the operands in *condition* is NULL. For example:

SELECT COUNT(*) FROM authors WHERE LNNVL(contract <> 1) -> 4

LOCALTIMESTAMP [(precision)]

Returns a *TIMESTAMP* value for the current date and time. This function is similar to *CURRENT_TIMESTAMP*, with the exception that this function does not return a *TIME_ZONE* value with the *TIMESTAMP*. For example:

SELECT LOCALTIMESTAMP FROM DUAL -> '15-APR-05 03.15.00 PM'

LOG(base_number, number)

Returns the logarithm of any *base_number* of *number*. For example:

SELECT LOG(50,10) FROM DUAL -> .58859191

LPAD(string1, number [,string2])

Returns *string1*, left-padded to length *number* using characters in *string2*; *string2* defaults to a single blank. For example:

SELECT LPAD('ucks',5,'d') FROM DUAL -> 'ducks'

LTRIM(string[, set])

Removes all characters in *set* from the left of *string*. *Set* defaults to a single blank. For example:

SELECT LTRIM(' Howdy! ',' ') FROM DUAL -> 'Howdy! '

MAKE_REF({table_name | view_name} , key [,...n])

Creates a reference (*REF*) to a row of an object view or a row in an object table whose object identifier is primary key-based.

MEDIAN(expression) OVER (partitioning)

Returns the median value in an ordered set of numeric or datetime values. See [Section 4.3](#) earlier in this chapter for a detailed explanation of the *partitioning* clause. For example:

SELECT MEDIAN(c1) FROM FIVE_NUMS -> 3

MONTHS_BETWEEN(date1, date2)

Returns the number of months between dates *date1* and *date2*. When *date1* is later than *date2*, the result is positive. If it is earlier, the result is negative. For example:

SELECT MONTHS_BETWEEN('15-APR-2000', '15-JUL-1999') FROM DUAL -> 9

NANVL(a, b)

Returns *b* when *a* is not a number (*NaN*); returns *a* otherwise. Expression *a* must evaluate to a *BINARY_FLOAT* or *BINARY_DOUBLE* number, which are the only number types that permit storing *NaN*. For example:

SELECT c1, NANVL(c1, 0) FROM NUMS

1.0E+000	1.0E+000
2.0E+000	2.0E+000
Nan	0

NCHAR(number)

A synonym for *CHR(number) USING NCHAR_CS*.

NEW_TIME(date, time_zone1, time_zone2)

Returns the date and time in *time_zone2* using *date* as the input date/time, and using *time_zone1* as the originating time zone. For example:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH12:MI:SS'
```

```
SELECT NEW_TIME(TO_DATE('04-15-99 08:22:31', 'MM-DD-YY HH12:MI:SS'),  
'AST', 'PST') FROM DUAL
```

```
15-APR-2099 04:22:31
```

Time_zone1 and *Time_zone2* may be any of these text strings:

AST, ADT: Atlantic Standard or Daylight Time

BST, BDT: Bering Standard or Daylight Time

CST, CDT: Central Standard or Daylight Time

EST, EDT: Eastern Standard or Daylight Time

GMT: Greenwich Mean Time

HST, HDT: Alaska-Hawaii Standard or Daylight Time

MST, MDT: Mountain Standard or Daylight Time

NST: Newfoundland Standard Time

PST, PDT: Pacific Standard or Daylight Time

YST, YDT: Yukon Standard or Daylight Time

NEXT_DAY(date, string)

Returns the date of the first weekday named by *string* that is later than *date*. The argument *string* must be either the full name or the abbreviation of a day of the week in the date language of the session. For example:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'
```

```
SELECT NEXT_DAY('15-APR-1999', 'SUNDAY') FROM DUAL
```

```
18-APR-1999
```

NLS_CHARSET_DECL_LEN(bytecnt, csid)

Returns the declaration width (*bytecnt*) of an *NCHAR* column using the character set ID (*csid*) of the column.

NLS_CHARSET_ID(text)

Returns the NLS character set ID number corresponding to *text*.

NLS_CHARSET_NAME(number)

Returns the *VARCHAR2* name for the NLS character set corresponding to the ID *number*.

NLS_INITCAP(string [, nlsparameter])

Returns *string* with the first letter of each word in uppercase and all other letters in lowercase. The *nlsparameter* offers special linguistic sorting features.

NLS_LOWER(string [, nlsparameter])

Returns *string* with all letters lowercase. The *nlsparameter* offers special linguistic sorting features.

NLSSORT(*string* [,*nlsparameter*])

Returns the string of bytes used to sort *string*. The *nlsparameter* offers special linguistic sorting features.

NLS_UPPER(*string* [,*nlsparameter*])

Returns *string* with all letters uppercase. The *nlsparameter* offers special linguistic sorting features.

NTILE(*expression*) *OVER* ([*partitioning*] *ordering*)

Divides an ordered data set into a number of groups numbered 1 to *expression* and assigns the appropriate group number to each row. Rows are allocated to each group so that the number of rows per group varies by no more than one. See [Section 4.3](#) earlier in this chapter for details on the *partitioning* and *ordering* clauses. For example:

SELECT c1, NTILE(4) OVER (ORDER BY c1) FROM FIVE_NUMS

```
1  1
2  1
3  2
4  3
5  4
```

NULLIF(*expr1*, *expr2*)

Returns NULL if *expr1* is equal to *expr2*. If the two expressions are not equal, then the function returns *expr1*. A NULL value, if passed as one of the expressions, can only be specified as the second expression *expr2*. For example:

SELECT c1, c2, NULLIF(c1, c2) FROM NUMS

```
1  2  1
2  2
3  2  3
```

NUMTODSINTERVAL(*number*, *string*)

Converts *number* to an *INTERVAL DAY TO SECOND* literal, where *number* is a number or an expression resolving to a number, such as a numeric datatype column. The second argument, *string*, can be 'DAY', 'HOUR', 'MINUTE', or 'SECOND', which specifies how to interpret *number*. For example:

SELECT NUMTODSINTERVAL(100, 'DAY') FROM DUAL

```
+000000100 00:00:00.000000000
```

NUMTOYMINTERVAL(*number*, *string*)

Converts *number* to an *INTERVAL YEAR TO MONTH* literal, where *number* is a number or an expression resolving to a number, such as a numeric datatype column. The second argument, *string*, can be 'YEAR' or 'MONTH', which specifies how to interpret *number*. For example:

SELECT NUMTOYMINTERVAL(100, 'YEAR') FROM DUAL

```
+000000100-00
```

NVL(*expression1*, *expression2*)

If *expression1* is NULL, *expression2* is returned in place of that NULL value. Otherwise, *expression1* is returned. *expression1* and *expression2* may be any datatype. For example:

```
SELECT NVL(2,29) FROM DUAL -> 2
```

NVL2(expression1, expression2, expression3)

Similar to *NVL*, except that if *expression1* is not NULL, *expression2* is returned. If *expression1* is NULL, *expression3* is returned. The expressions may be any datatype except *LONG*. For example:

```
SELECT NVL2(1,3,5) FROM DUAL -> 3
```

ORA_HASH(expression [, buckets [, seed]])

Computes a hash value from *expression* and returns a bucket number based on the computed hash value. The optional *buckets* argument is the maximum bucket number to use, which is one less than the total number of buckets, since the bucket numbering starts at 0. The default for *buckets* is 4,294,967,295. The optional *seed* value is used to seed the hashing function so that multiple results can be produced from the same data by only changing the *seed* value. The *seed* value defaults to 0. This example psuedo-randomly assigns all numbers to one of two buckets and returns those assigned to the first bucket, which will be a sample of roughly one half of the values:

```
SELECT C1 FROM FIVE_NUMS WHERE
```

```
ORA_HASH( C1, 1, TO_CHAR( SYSTIMESTAMP, 'SSSS.FF' ) ) = 0
```

```
1
```

```
5
```

PATH(number)

Returns the path specified by the *UNDER_PATH* condition with the correlation variable *number* in an XML query. See the Oracle SQL Reference for more information.

POWERMULTISET(nested_table) POWERMULTISET_BY_CARDINALITY(nested_table, cardinality)

Returns a nested table of nested tables of all nonempty subsets of the input nested table in the *nested_table* parameter. *POWERMULTISET_BY_CARDINALITY* has an additional parameter that can be used to limit the subsets returned to a specified minimum cardinality. For more information, please see the Oracle SQL Reference.

PRESENTNNV(cell_reference, expr1, expr2)

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns *expr1* when *cell_reference* exists and is not NULL; otherwise, it returns *expr2*.

PRESENTV(cell_reference, expr1, expr2)

Relevant only in the interrow calculations performed within the *MODEL* clause of a *SELECT* statement. This function returns *expr1* when *cell_reference* exists; otherwise, it returns *expr2*.

PREVIOUS(cell_reference)

Relevant only in the inter-row calculations performed within the *ITERATE...[UNTIL]* section of a *SELECT*'s *MODEL* clause. This function returns the value held by *cell_reference* at the beginning of the iteration.

RATIO_TO_REPORT (value_exprs) OVER (partitioning)

Computes the ratio of a value in *value_exprs* to the sum of all *value_exprs* with each partition. If *values_expr* is NULL, the ratio-to-report value also is NULL. See [Section 4.3](#) earlier in this chapter for details on the *partitioning* clause. For example:


```
SELECT c1, RATIO_TO_REPORT(c1) OVER ( ) FROM FIVE_NUMS
```

```
1      .066666667
2      .133333333
3      .2
4      .266666667
5      .333333333
```

RAWTOHEX(*raw*)

Converts a *raw* value to a string (character datatype) of its hexadecimal equivalent. For example:

```
SELECT RAWTOHEX('Hi') FROM DUAL -> 4869
```

RAWTONHEX(*raw*)

Converts a *raw* value to an *NVARCHAR2* (character datatype) of its hexadecimal equivalent.

REF(*table_alias*)

REF takes a table alias associated with a row from an object table or an object view. A special reference value is returned for the object instance that is bound to the variable or row.

REFTOHEX(*expression*)

Converts *expression* to a character value containing its hexadecimal equivalent.

REGEXP_INSTR(*string*, *pattern* [, *start_at* [, *occurrence* [, *roption* [, *mparam*]]])

Returns the character position within *string* matching the regular expression pattern in *pattern*. **REGEXP_INSTR** searches *string* with a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *roption* parameter can be 0 or 1 and specifies if the position returned is the first character matching the pattern or the character after. The default for *roption* is 0, which returns the position of the first character matching the pattern. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

'i'

matching is case-insensitive

'c'

matching is case-sensitive

'n'

the "." character matches newline characters

'm'

treat input string as multiple lines; use '^' to match the beginning of a line and '\$' to match the end of a line

For example:

```
SELECT REGEXP_INSTR( 'Hello, World!', '([ ^ ]*)!', 1, 1) FROM DUAL -> 8
```

REGEXP_REPLACE(*string*, *pattern* [, *newstr* [, *start_at* [, *occurrence* [, *mparam*]]]])

Returns the result of replacing all occurrences of the regular expression in *pattern* within *string* with another string *newstr*. **REGEXP_REPLACE** searches *string* with a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

'i'

matching is case-insensitive

'c'

matching is case-sensitive

'n'

the "." character matches newline characters

'm'

treat input string as multiple lines; use '^' to match the beginning of a line and '\$' to match the end of a line

For example:

```
SELECT REGEXP_REPLACE( 'Hello, World!', '([^\ ]*!)', 'Reader!') FROM DUAL
```

```
'Hello, Reader!'
```

REGEXP_SUBSTR(*string*, *pattern* [, *start_at* [, *occurrence* [, *mparam*]]])

Returns the substring within *string* matching the regular expression pattern *pattern*. **REGEXP_SUBSTR** searches *string* with a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

'i'

matching is case-insensitive

'c'

matching is case-sensitive

'n'

the '.' character matches newline characters

'm'

treat input string as multiple lines; use '^' to match the beginning of a line and '\$' to match the end of a line

For example:

```
SELECT REGEXP_SUBSTR( 'Hello, World!', '([^\ ]*!)' ) FROM DUAL
```

```
'World!'
```

REMAINDER(*m*, *n*)

Returns the remainder of *m* divided by *n*. This return value is equivalent to the expression:

$m-n*\text{ROUND}(m/n)$

The function *MOD* uses *FLOOR* instead of *ROUND*. For example:

SELECT REMAINDER(11, 4), MOD(11, 4) FROM DUAL

-1 3

REPLACE(string, search_string [,replacement_string])

Returns *string* with every occurrence of *search_string* replaced with *replacement_string*. For example:

SELECT REPLACE('change', 'e', 'ing') FROM DUAL -> 'changing'

ROUND (number[, decimal])

Returns *number* rounded to *decimal* places right of the decimal point. When *decimal* is omitted, *number* is rounded to an integer. Note that *decimal*, an integer, can be negative to round off digits left of the decimal point. For example:

SELECT ROUND(12345.6789, 2) FROM DUAL -> 12345.68

ROUND (date[, format])

Returns the *date* rounded to the unit specified by the *format* model. When *format* is omitted, date is rounded to the nearest day. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

SELECT ROUND(TO_DATE('15-APR-1999'), 'MONTH') FROM DUAL

01-APR-1999

ROWIDTOCHAR(rowid), ROWIDTONCHAR(rowid)

ROWIDTOCHAR converts a *rowid* value to the *VARCHAR2* datatype, 18 characters long. *ROWIDTONCHAR* converts *rowid* to an 18-character long *NVARCHAR2* value. For example:

SELECT ROWIDTOCHAR(ROWID) FROM NUMS

ABAsxDAAKAAAEEqAAA

ABAsxDAAKAAAEEqAAB

ABAsxDAAKAAAEEqAAC

ABAsxDAAKAAAEEqAAD

RPAD(string1, number [, string2])

Returns *string1*, right-padded to length *number* with the value of *string2*, repeated as needed. *string2* defaults to a single blank. For example:

SELECT RPAD('duck',8,'s') FROM DUAL -> 'duckssss'

RTRIM(string[,set])

Returns *string*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. For example:

SELECT RTRIM(' welcome ', ' ') FROM DUAL -> ' welcome'

SCN_TO_TIMESTAMP(scn)

Returns the timestamp associated with the System Change Number (SCN) argument. For example:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM NUMS WHERE c1 = 1
```

```
15-APR-04 02.56.05.000000000 PM
```

SESSIONTIMEZONE

Returns the session's time zone offset. For example:

```
SELECT SESSIONTIMEZONE FROM DUAL -> -06:00
```

SET(nested_table)

Returns a nested table of distinct elements from the input nested table. For more information, please see the Oracle SQL Reference.

SIGN(number)

When *number* < 0, returns -1. When *number* = 0, returns 0. When *number* > 0, returns 1. For example:

```
SELECT SIGN(-3.1415926) FROM DUAL -> -1
```

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

SINH(number)

Returns the hyperbolic sine of *number*. For example:

```
SELECT SINH(180) FROM DUAL -> 7.4469E+77
```

SOUNDEX(string)

Returns a character string containing the phonetic representation of *string*. This function allows words that are spelled differently but sound alike in English to be compared for equality. For example:

```
SELECT SOUNDEX('thimble') FROM DUAL -> 'T514'
```

STATS_BINOMIAL_TEST, STATS_CROSSTAB, STATS_F_TEST, STATS_KS_TEST, STATS_MODE, STATS_MW_TEST, STATS_ONE_WAY_ANOVA, STATS_T_TEST_ONE, STATS_T_TEST_PAired, STATS_T_TEST_INDEP, STATS_T_TEST_INDEPU, STATS_WSR_TEST

Oracle provides many sophisticated statistical functions. For further information on the *STATS_** functions, please see the Oracle SQL Reference.

STDDEV([DISTINCT | ALL] expression) [OVER (window_clause)]

Returns a sample standard deviation of a set of numbers shown as *expression*. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV(col1) FROM NUMS -> 5.71547607
```

STDDEV_SAMP(*expression*) [*OVER* (*window_clause*)]

Computes the cumulative sample standard deviation and returns the square root of the sample variance. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV_SAMP(col1) FROM NUMS    -> 5.71547607
```

STDEV_POP(*expression*) [*OVER* (*window_clause*)]

Computes the population standard deviation and returns the square root of the population variance. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV_POP(col1) FROM NUMS    -> 4.94974747
```

SUBSTR(*string*, *start* [*FROM starting_position*] [*FOR length*])

Refer to the earlier section on *SUBSTR*. For example:

```
SELECT SUBSTR('Hello, World!',8,10) FROM DUAL -> 'World!'
```

SUBSTRB(*extraction_string* [, *length*])

Returns the portion of *string* beginning at *start* and continuing for *length* characters. If *length* is omitted, all characters from *start* onward are returned. If *start* is negative, it represents an offset from the right edge of the *string*. For example:

```
SELECT SUBSTR ('Hello, World!', 8)
```

```
FROM DUAL -> World!
```

Use *SUBSTRB* for bytes, *SUBSTRC* for UNICODE complete characters, *SUBSTR2* for UNICODE UCS2 code points, and *SUBSTR4* for UNICODE UCS4 code points.

SYS_CONNECT_BY_PATH(*column*, *char*)

For hierarchical queries, *SYS_CONNECT_BY_PATH* returns the path from the root to the node with the column name specified by the *column* parameter. The *char* parameter specifies the node separator for the return path. For more on Oracle hierarchical queries, please refer to the Oracle SQL Reference.

SYS_CONTEXT(*namespace*, *attribute* [, *length*])

Returns the value of *attribute* associated with the context *namespace*, usable in both SQL and PL/SQL statements. For example:

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL
```

```
'LOGIN'
```

SYS_DBURIGEN(*column* [, *rowid*] [, ...] [, 'text() '])

Returns a URL that can be used as a unique reference to the row specified by the *column* parameter. For columns that don't hold unique values, a *rowid* can be used directly after the column it identifies to guarantee that the URL only points to one row. Use the 'text()' option if you want the URL to point to the text within an XML document, instead of the document itself.

SYS_EXTRACT_UTC(*datetime*)

Returns the *datetime* argument converted to a UTC datetime value. For example:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2004-04-15 11:59:59.00 -08:00')  
FROM DUAL  
  
'15-APR-04 07.59.59.000000000 PM'
```

SYS_GUID()

Generates and returns a globally unique identifier (*RAW* value) made up of 16 bytes. For example:

```
SELECT SYS_GUID( ) FROM DUAL -> C0FD3FDC30148EAE030440A49096A41
```

SYS_TYPEID(object_value)

Returns the type ID of the *object_value* parameter.

SYS_XMLAGG(expr [, format])

Returns a single XML document created by aggregating the XML documents or fragments in the *expr* parameter. The optional *format* parameter can be used to format the XML document. For more information, refer to the Oracle SQL Reference.

SYS_XMLGEN(expression [, format])

Returns a single XML document created from the *expression*. The optional *format* parameter can be used to format the XML document. For more information, refer to the Oracle SQL Reference.

SYSDATE

Returns the current date and time on the system in which the database is hosted. The value returned is of type *DATE*. For example:

```
SELECT SYSDATE FROM DUAL -> 26-JUN-2003
```

SYSTIMESTAMP

Returns the current date and time on the system in which the database is hosted. The value returned is of type *TIMESTAMP*. For example:

```
SELECT SYSTIMESTAMP FROM DUAL  
  
26-JUN-2003 11.15.00.000000 PM -06:00
```

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) FROM DUAL -> -0.000093
```

TANH(number)

Returns the hyperbolic tangent of *number*. For example:

```
SELECT TANH(180) FROM DUAL -> 1
```

TIMESTAMP_TO_SCN(timestamp_value)

Returns the approximate system change number (*SCN*) associated with that timestamp. The return value is of

type *NUMBER*.

TO_BINARY_DOUBLE (*expr* [, *format* [, *nls_parameter*]])

Converts *expr* to a *BINARY_DOUBLE* in the format specified by the *format* parameter. If *expr* is a character expression, the *format* and *nls_parameter* options have equivalent meanings, as they do in the *TO_CHAR* function. If *expr* is a numeric expression, then *format* and *nls_parameter* must be omitted. For example:

SELECT c1, TO_BINARY_DOUBLE(c1) FROM NUMS

1	1.0E+000
2	2.0E+000
3	3.0E+000

TO_BINARY_FLOAT(*expr* [, *format* [, *nls_parameter*]])

Converts *expr* to a *BINARY_FLOAT* in the format specified by the *format* parameter. If *expr* is a character expression, the *format* and *nls_parameter* options have equivalent meanings as they do in the *TO_CHAR* function. If *expr* is a numeric expression, then *format* and *nls_parameter* must be omitted. For example:

SELECT c1, TO_BINARY_FLOAT(c1) FROM NUMS

1	1.0E+000
2	2.0E+000
3	3.0E+000

TO_CHAR (*character_expr*)

Converts *character_expr* to the database character set. For example:

SELECT TO_CHAR('Howdy') FROM DUAL

Howdy

TO_CHAR (*date* | *interval* [, *format* [, *nls_parameter*]])

Converts *date* or *interval* to a *VARCHAR2* in the format specified by the date *format*. When *format* is omitted, *date* is converted to the default date format. The *nls_parameter* option offers additional control over formatting. Listed below are the available specifiers for *format* and their meanings.

AD or *A.D.*

AD indicator

AM or *A.M.*

Meridian indicator

BC or *B.C.*

BC indicator

D

Day of week (1-7)

DAY

Name of day

DD

Day of month (1-31)

DDD

Day of year (1-366)

DL

Long date format

DS

Short date format

DY

Abbreviated name of day

FF

Fractional seconds; to specify the precision, include a number (1-9) after the FF specifier

HH or HH12

Hour of day (1-12)

HH24

Hour of day (0-23)

J

Julian day; the number of days since January 1, 4713 BC

MI

Minute (0-59)

MM

Month (01-12)

MON

Abbreviated name of month

RM

Roman numeral month (I-XII)

SS

Second (0-59)

SSSSS

Seconds past midnight (0-86,399)

SYYY

Four-digit year; BC dates are prefixed with a minus sign

TS

Short time format.

TZD

Daylight savings information (example: PST versus PDT)

TZH

Time zone hour

TZM

Time zone minute

TZR

Time zone region

X

Local radix character

Y, YY, or YYY

One, two, or three digits for the year

Y,YYY

Year with comma

YYYY

Four-digit year

For example:

```
SELECT TO_CHAR(TO_DATE('15-APR-1999'), 'MON-DD-YYYY') FROM DUAL
```

```
APR-15-1999
```

TO_CHAR (number [, format [, nls_parameter]])

Converts *number* to a *VARCHAR2* in the *format* specified. When *format* is omitted, *number* is converted to a string long enough to hold the number. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_CHAR(123.45, '$999.99') FROM DUAL -> $123.45
```

TO_CLOB (expr)

Converts the character expression given by *expr* to the *CLOB* datatype. For example:

```
SELECT LENGTH(TO_CLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_DATE(string [, format [, nls_parameter]])

Converts *string* (in *CHAR* or *VARCHAR2*) to a *DATE* datatype. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_DATE('15/04/1999', 'DD/MM/YYYY') FROM DUAL  
  
15-APR-1999
```

TO_DSINTERVAL(string [, nls_parameter])

Converts the character expression given by *string* to the *INTERVAL DAY TO SECOND* datatype. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT CURRENT_DATE, CURRENT_DATE-TO_DSINTERVAL('14 00:00:00') FROM DUAL  
  
15-APR-2003    01-APR-2003
```

TO_LOB(long_column)

Usable only by *LONG* or *LONG RAW* expressions, it converts *LONG* or *LONG RAW* values in the column *long_column* to *LOB* values. It is usable only in the *SELECT* list of a subquery in an *INSERT* statement.

TO_MULTI_BYTE(string)

Returns *string* with all of its single-byte characters converted to their corresponding multibyte characters.

TO_NCHAR(expr [, format [, nls_parameter]])

Synonymous with the *TO_CHAR* function, except the return datatype is *NCHAR*. For more on valid format specifiers, see the *TO_CHAR* function.

TO_NCLOB (expr)

Converts the character expression given by *expr* to the *NCLOB* datatype. For example:

```
SELECT LENGTH(TO_NCLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_NUMBER(string [, format [, nls_parameter]])

Converts a numeric *string* (of *CHAR* or *VARCHAR2* datatype) to a value of *NUMBER* datatype, optionally in the format specified by the *format* model. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_NUMBER('12345') FROM DUAL    -> 12345
```

TO_SINGLE_BYTE(string)

Returns *string* with all of its multibyte characters converted to their corresponding single-byte characters.

TO_TIMESTAMP(string [, format [, nls_parameter]])

Converts the character expression provided by *string* to the *TIMESTAMP* datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TO_TIMESTAMP(CURRENT_DATE) FROM DUAL -> 04-MAY-04 12.00.00 AM
```

TO_TIMESTAMP_TZ(*string* [, *format* [, *nls_parameter*]])

Converts the character expression provided by *string* to the *TIMESTAMP WITH TIME ZONE* datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TO_TIMESTAMP_TZ('15-04-2006', 'DD-MM-YYYY') FROM DUAL
```

```
15-APR-06 12.00.00.000000000 AM -07:00
```

TO_YMINTERVAL(*string*)

Converts the character expression provided by *string* to the *INTERVAL YEAR TO MONTH* datatype. For example:

```
SELECT TO_DATE('29-FEB-2000')+TO_YMINTERVAL('04-00') FROM DUAL
```

```
29-FEB-04
```

TRANSLATE(*char_value*, *from_text*, *to_text*)

Returns *char_value* with each occurrence of a character in *from_text* replaced by its corresponding character in *to_text*. For example:

```
SELECT TRANSLATE('foobar', 'fa', 'bu') FROM DUAL -> 'boobur'
```

TRANSLATE (*text* USING [*CHAR_CS* | *NCHAR_CS*])

Converts *text* into the character set specified. Use *CHAR_CS* to convert *text* to the *CHAR* datatype or *NCHAR_CS* to convert *text* to the *NCHAR* datatype. For example:

```
SELECT TRANSLATE(N'foobar' USING CHAR_CS) FROM DUAL
```

```
'foobar'
```

TREAT (*expr* AS [*REF*] [*schema.*]*type*)

Converts *expr* from its declared type to the type specified in the *type* parameter. For more information on the usage of this function, please look to the SQL Reference for the Oracle Database.

TRUNC (*base* [, *number*])

Returns *base* truncated to *number* decimal places. When *number* is omitted, *base* is truncated to an integer. *number* can be negative to truncate (make zero) digits left of the decimal point. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TRUNC('123.456', 2) FROM DUAL -> 123.45
```

TRUNC (*date* [, *format*])

Returns *date* truncated to the unit specified by *format*. When *format* is omitted, *date* is truncated to the nearest whole day. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT TRUNC(TO_DATE('15/04/1999', 'MM/DD/YYYY'), 'YYYY')
```

FROM DUAL

1999

See *TO_CHAR* for a list of format specifiers.

TZ_OFFSET ({ *expr* | *SESSIONTIMEZONE* | *DBTIMEZONE* }

Returns the time zone offset corresponding to the argument. The character expression *expr* can either be the name of the time zone or a time zone offset. The *SESSIONTIMEZONE* and *DBTIMEZONE* arguments provide the time zone for the session or database, respectively. For example:

SELECT TZ_OFFSET('+08:00'), TZ_OFFSET(SESSIONTIMEZONE) FROM DUAL

+08:00 -07:00

UID

Returns an integer that uniquely identifies the currently logged on session user. No parameters are needed. For example:

SELECT UID FROM DUAL -> 47

UNISTR(*string*)

Converts *string* to the *NCHAR* datatype while converting any UNICODE encoding values within *string*. For example:

SELECT UNISTR('Ei Ni\00F1o') FROM DUAL -> 'El Niño'

UPDATEXML(*instance*, *xpath*, *expr* [, *namespace*]

Updates the values held by nodes within *instance* to the new value in *expr*. Only those nodes returned by the XPath query contained in the *xpath* parameter are updated. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

SELECT UPDATEXML(XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),

'/foo/bar', '<bar>Bye, World!</bar>') from DUAL

Bye, World!

USERENV(*option*)

Returns information about the current session in *VARCHAR2*. This function has been deprecated and is only provided for backward compatibility. *USERENV* is a synonym for *SYS_CONTEXT*('USER_ENV', *option*). Look to the *SYS_CONTEXT* function with the *USERENV* namespace for current functionality. For example:

SELECT USERENV('LANGUAGE') "Language" FROM DUAL

'AMERICAN_AMERICA.AL32UTF8'

VALUE(*table_alias*)

Takes a *table_alias* associated with a row in an object table and returns the object instance stored within the object table for that row.

VARIANCE([*DISTINCT*] *expression*) [*OVER* (*window_clause*)]

Returns the variance of *expression* calculated as follows: 0 if the number of rows in *expression* = 1; *VAR_SAMP* if the number of rows in *expression* > 1. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For

example:

```
SELECT VARIANCE(col1) FROM NUMS -> 32.6666667
```

VAR_POP(*expression*) [*OVER* (*window_clause*)]

Returns the population variance of the set of numbers represented by *expression* after discarding the NULLs in the set. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT VAR_POP(col1) FROM NUMS -> 24.5
```

VAR_SAMP(*expression*) [*OVER* (*window_clause*)]

Returns the sample variance of the set of numbers represented by *expression* after discarding the NULLs in the set. See [Section 4.3](#) earlier in this chapter for details on the *window_clause*. For example:

```
SELECT VAR_SAMP(col1) FROM NUMS -> 32.6666667
```

VSIZE(*expression*)

Returns the number of bytes in the internal representation of *expression*. When *expression* is *NULL*, returns *NULL*. For example:

```
SELECT vsize(1) FROM DUAL -> 2
```

XMLAGG(*instance* [, *order_by*])

An aggregate function that returns an XML document from a table of XML fragments contained in *instance*. The optional *order_by* clause allows the XML fragments in the result to be ordered. For more information on XML queries refer to the Oracle SQL Reference.

XMLCOLATTVAL(*expr* [*AS alias*] [,...])

Returns an XML fragment from the *expr* arguments. The optional *AS* clause can be used to change the value of the name attribute. For more information on XML queries, refer to the Oracle SQL Reference.

XMLCONCAT(*instance* [,...])

Returns an XML instance that is the union of all XML *instance* parameters. For more information on XML queries refer to the Oracle SQL Reference.

XMLELEMENT([*NAME*] *name* [, **XMLATTRIBUTES**(*expr* [*AS alias*] [,...])] [, *value* [,...]])

Returns an **XMLELEMENT** with the name specified in the *name* parameter and attributes specified in the optional **XMLATTRIBUTES** clause. The *value* parameters provide the values of the **XMLELEMENT** result. For more information on XML queries refer to the Oracle SQL Reference.

XMLFOREST(*value* [*AS alias*] [,...])

Returns an XML fragment constructed from the values provided in the *value* parameters. The optional *AS* clause can be used to change the enclosing tag name. For more information on XML queries refer to the Oracle SQL Reference.

XMLSEQUENCE(*instance*)

Returns an array of XML fragments constructed from the top-level nodes in the XML instance provided by the *instance* parameter. For more information on XML queries, refer to the Oracle SQL Reference.

XMLTRANSFORM(*instance*, *stylesheet*)

Returns the result of applying the XSL stylesheet in the *stylesheet* parameter to the XML document contained in *instance*. For more information on XML queries, refer to the Oracle SQL Reference.

4.5.4 PostgreSQL-Supported Functions

This section lists the functions specific to PostgreSQL, with examples and corresponding results.

ABSTIME(timestamp)

Converts the *timestamp* value to *ABSTIME* type. This function is provided for backwards compatibility and may be removed in future versions. For example:

```
SELECT ABSTIME(CURRENT_TIMESTAMP) -> 2003-06-24 00:19:17-07
```

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

AGE(timestamp)

Same meaning as *AGE(CURRENT_DATE, timestamp)*.

AGE(timestamp, timestamp)

Returns the time between the two *timestamp* values. For example:

```
SELECT AGE( '2003-12-31', CURRENT_TIMESTAMP )
```

```
6 mons 7 days 00:34:41.658325
```

AREA(object)

Returns the area of an item. For example:

```
SELECT AREA( BOX '((0,0),(1,1))' ) -> 1
```

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Returns the arctangent of *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(float1, float2)

Returns the arctangent of the two *float* values. *ATAN2(x,y)* is similar to *ATAN(y/x)*, with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

SELECT ATAN2(3.1415926, 0) -> 1.5707963267949

BOX(box, box)

Returns a *BOX* created by the intersection of the two boxes. If the two boxes do not intersect, the return value is NULL. For example:

SELECT BOX(BOX '((-1,-1),(1,1))', BOX '((0,0),(1,1))')

(1,1),(0,0)

BOX(circle)

Returns a *BOX* with vertices that intersect *circle* so that the box is the maximum size that can be fully contained within *circle*. For example:

SELECT BOX(CIRCLE '((0,0),2.0)')

(1.41421356237309,1.41421356237309),

(-1.41421356237309,-1.41421356237309)

BOX(point, point)

Returns a *BOX* with the two point arguments as opposite corners. For example:

SELECT BOX(POINT(0,0), POINT(1,1)) -> (1,1),(0,0)

BOX(polygon)

Converts *polygon* to a *BOX*. For example:

SELECT BOX(POLYGON '((0,0),(1,1),(2,0))') -> (2,1),(0,0)

BROADCAST(inet)

Constructs a broadcast address as text. For example:

SELECT BROADCAST('192.168.1.5/24') -> '192.168.1.255/24'

CBRT(float8)

Returns the cube root of *float8*. For example:

SELECT CBRT(8) -> 2

CENTER(object)

Returns a *POINT* object to the center of the argument. For example:

SELECT CENTER(CIRCLE '((0,0), 2.0)') -> (0,0)

CHAR(text)

Converts *text* to the *CHAR* type.

CHAR_LENGTH(string) or *CHARACTER_LENGTH(string)*

Returns the length of *string* in characters.

CIRCLE(box)

Returns a *CIRCLE* contained within *box*. For example:

SELECT CIRCLE(BOX '((0,0),(1,1)')) -> <(0.5,0.5),0.707106781186548>

CIRCLE(point, float8)

Converts *point* to a *CIRCLE* with *float8* for the radius. For example:

SELECT CIRCLE(POINT '(0,0)', 2.0) -> <(0,0),2>

COALESCE(list)

Returns the first non-NULL value in *list*. For example:

SELECT COALESCE(NULL,1,2,3,NULL) -> 1

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

COT(number)

Returns the cotangent of *number*. For example:

SELECT COT(3.1415) -> -10792.88993953

DATE_PART(text, value)

Equivalent to *EXTRACT(text, value)*; for more details on the usage of *EXTRACT*, see the section about *EXTRACT* in [Section 4.4](#).

DATE_TRUNC(precision, timestamp)

Truncates *timestamp* to the specified *precision*. For example:

SELECT DATE_TRUNC('hour', TIMESTAMP '2003-04-15 23:58:30')

2003-04-15 23:00:00

DEGREES(float8)

Converts radians to degrees. For example:

SELECT DEGREES(3.1415926) -> 179.999996929531

DIAMETER(circle)

Returns the diameter of *circle*. For example:

SELECT DIAMETER(CIRCLE(POINT '(0,0)', 2.0)) -> 4

FLOAT(int)

Converts *int* to a floating point.

FLOAT4(int)

Converts *int* to a floating point.

HEIGHT(box)

Returns the vertical size of *box*. For example:

SELECT HEIGHT(BOX '((0,0),(1,1))) -> 1

HOST(inet)

Extracts the host address as text. For example:

SELECT HOST('192.168.1.5/24') -> '192.168.1.5'

INITCAP(text)

Converts the first letter of each word to uppercase. For example:

SELECT INITCAP('my name is inigo montoya.')

'My Name Is Inigo Montoya.'

INTEGER(float)

Converts a floating point to integer.

INTERVAL(reltime)

Converts *reltime* to an *INTERVAL*.

ISCLOSED(path)

Returns 't' if *path* is closed, 'f' if open. For example:

SELECT ISCLOSED(PATH '((0,0),(1,1),(2,0))) -> 't'

SELECT ISCLOSED(PATH '[(0,0),(1,1),(2,0)]') -> 'f'

ISFINITE(interval)

Returns 'f' if *interval* is open, 't' otherwise. For example:

SELECT ISFINITE(INTERVAL '4 hours') -> 't'

ISFINITE(timestamp)

Returns 'f' if *timestamp* is either invalid or infinite, 't' otherwise. For example:

```
SELECT ISFINITE(TIMESTAMP '2001-02-16 21:28:30') -> 't'
```

ISOPEN(path)

Returns an open *path*. For example:

```
SELECT ISOPEN(PATH '((0,0),(1,1),(2,0)))' -> 'f'
```

```
SELECT ISOPEN(PATH '[(0,0),(1,1),(2,0)]' -> 't'
```

LENGTH(object)

Returns the length of *object*. For example:

```
SELECT LENGTH('Howdy!') -> 6
```

```
SELECT LENGTH(PATH '((-1,0),(1,0))' -> 4
```

LOG(float8)

Returns a base-10 logarithm. For example:

```
SELECT LOG( 100 ) -> 2
```

LPAD(text,int,text)

Returns a left-pad string to the specified length. For example:

```
SELECT LPAD('Duck', 10, 's') -> 'sssssDuck'
```

LSEG(box)

Converts a *box* diagonal to a line segment. For example:

```
SELECT LSEG(BOX '((-1,0),(1,0))' -> [(1,0),(-1,0)]
```

LSEG(point,point)

Converts *points* to a line segment. For example:

```
SELECT LSEG(POINT '(-1,0)', POINT '(1,0)') -> [(-1,0),(1,0)]
```

LTRIM(text)

Returns *text* with all leading whitespace removed. For example:

```
SELECT LTRIM(' Howdy! ') -> 'Howdy! '
```

MASKLEN(cidr)

Returns the netmask length in *cidr*. For example:

```
SELECT MASKLEN('192.168.1.5/24') -> 24
```

NETMASK(inet)

Returns the netmask for *inet*. For example:

```
SELECT NETMASK('192.168.1.5/24') -> '255.255.255.0'
```

NETWORK(inet)

Returns the network part of *inet*. For example:

```
SELECT NETWORK('192.168.1.5/24') -> '192.168.1.0/24'
```

NPOINTS(object)

Returns the number of points in *object*. For example:

```
SELECT NPOINTS(POLYGON '((1,1),(0,0)')) -> 2
```

NULLIF(input, value)

Returns NULL if *input* = *value*, else returns *input*. For example:

```
SELECT NULLIF( 5, 6 ), NULLIF( 5, 5 )
```

```
5 NULL
```

PATH(polygon)

Converts *polygon* to a path. For example:

```
SELECT PATH( '((0,0),(1,1),(2,0))' )
```

```
((0,0),(1,1),(2,0))
```

PCLOSE(path)

Converts *path* to a closed path. For example:

```
SELECT PCLOSE(PATH '['(0,0),(1,1),(2,0)'])
```

```
((0,0),(1,1),(2,0))
```

PI()

Returns the constant pi.

POLYGON(path)

Converts *path* to a *POLYGON*. For example:

```
SELECT POLYGON(PATH '['(0,0),(1,1),(2,0)'])
```

```
((0,0),(1,1),(2,0))
```

POINT(circle)

Returns the center of *circle*. For example:

SELECT POINT(CIRCLE '((0,0), 2.0)') -> (0,0)

POINT(lseg1,lseg2)

Returns the intersection of two line segments. For example:

SELECT POINT(LSEG '((-1,0),(1,0))', LSEG '((-2,-2),(2,2))')
(0,0)

POINT(polygon)

Returns the center point of *polygon*. For example:

SELECT POINT(POLYGON '((0,0),(1,1),(2,0))')
(1,0.3333333333333333)

POLYGON(box)

Returns a 4-point polygon. For example:

SELECT POLYGON(BOX '((0,0),(1,1))')
((0,0),(0,1),(1,1),(1,0))

POLYGON(circle)

Synonym for *POLYGON(12, circle)*.

POLYGON(npts,circle)

Returns and approximation of *circle* as a polygon with *npts* vertices. For example:

SELECT POLYGON(6, CIRCLE '((0,0),2.0)')
((-2,0),
(-0.9999999999999999,1.73205080757228),
(1.00000000001179,1.73205080756207),
(2,-2.04136478690279e-11),
(0.999999999976428,-1.73205080758249),
(-1.00000000002946,-1.73205080755187))

POPEN(path)

Converts *path* to an open path. For example:

SELECT POPEN(PATH '((0,0),(1,1),(2,0))')
[[0,0),(1,1),(2,0]]

POW (number,exponent)

Raises a *number* to the specified *exponent*. For example:

SELECT POW(2, 3) -> 8

RADIANS(float8)

Converts degrees to radians. For example:

SELECT RADIANS(180) -> 3.14159265358979

RADIUS(circle)

Returns the radius of *circle*. For example:

SELECT RADIUS(CIRCLE '((0,0), 2.0)') -> (0,0)

RELTIME(interval)

Converts *interval* to a *RELTIME*. Provided for backward compatibility and may be removed in a future release.

ROUND(number)

Rounds *number* to the nearest integer. For example:

SELECT ROUND(5.5) -> 6

RPAD(text,length,char)

Pads *text* to the specified *length* using *char*. For example:

SELECT RPAD('Duck', 10, 's') -> 'Duckssssss'

RTRIM(text)

Returns *text* with all trailing whitespace removed. For example:

SELECT RTRIM(' St. Lucia ') -> ' St. Lucia'

SET_MASKLEN(inet, size)

Sets the netmask length for *inet* to *size*. For example:

SELECT SET_MASKLEN('192.168.1.5/24',16)

'192.168.1.5/16'

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SUBSTRING(string [FROM start] [FOR bytes]), SUBSTR(string,start[,bytes])

Extracts a substring of length *bytes* from *string* starting at the character position *start*. If *bytes* is omitted, then the length returned is the remainder of the *string* from the *start* position. For example:

SELECT SUBSTRING('Inigo Montoya' FROM 7 FOR 4) -> 'Mont'

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

SELECT TAN(3.1415) -> -0.000093

TEXT(char)

Converts *char* to *TEXT* type.

TIMESTAMP(date [,time])

Converts *date* to a timestamp.

TO_CHAR(expression, text)

Converts *expression* to a string. For example:

SELECT TO_CHAR(NUMERIC '-125.8', '999D99S') -> 125.80-

SELECT TO_CHAR (interval '15h 2m 12s','HH24:MI:SS') -> 15:02:12

TO_DATE(string, format)

Converts *string* to a date using the second argument for the input format. Listed below are the available specifiers for *format* and their meanings.

AM or A.M.

Meridian indicator

AD or A.D.

AD indicator

BC or B.C.

BC indicator

CC

Two-digit century

D

Day of week (1-7)

DD

Day of month (01-31)

DDD

Day of year (001-366)

DAY

Full uppercase day name

Day

Full camel case day name

day

Full lowercase day name

DY

Abbreviated uppercase day name

Dy

Abbreviated camel case day name

dy

Abbreviated lowercase day name

HH or HH12

Hour of day (01-12)

HH24

Hour of day (00-23)

IW

ISO week number of year

J

Julian day number (days since January 1, 4713 BC)

MI

Minute of hour (00-59)

MM

Two-digit month number (01-12)

MON

Abbreviated uppercase month name

Mon

Abbreviated camel case month name

mon

Abbreviated lowercase month name

MONTH

Full uppercase month name

Month

Full camel case month name

month

Full lowercase month name

MS

Milliseconds (000-999)

PM or P.M.

Meridian indicator

Q

Quarter of the year

RM

Roman Numeral month (I-XII)

rm

Lowercase Roman Numeral month (i-xii)

SS

Seconds (00-59)

SSSS

Seconds past midnight (0-86399)

TZ

Uppercase time zone name

tz

Lowercase time zone name

US

Microseconds (000000-999999)

W

Week of month (1-5)

WW

Week of year (1-53)

Y, YY, or YYY

One-, two-, or three-digit years

Y,YYYY

Year with comma

YYYY

Four-digit year

For example:

```
SELECT TO_DATE('05 Dec 2000', 'DD Mon YYYY') -> 2000-12-05
```

TO_NUMBER(string, format)

Converts *string* to a numeric value using the second argument for the input format. For example:

```
SELECT TO_NUMBER('12,454.8-', '99G999D9S') -> -12454.8
```

TO_TIMESTAMP(text, format)

Converts *text* to a timestamp value using the second argument for the input format. (For more on valid format specifiers, see the *TO_DATE* function.) For example:

```
SELECT TO_TIMESTAMP('05 Dec 2000', 'DD Mon YYYY') -> 2000-12-05 00:00:00-08
```

TRANSLATE(text,from,to)

Converts the characters found in *text* that also exist in *from* to the corresponding characters in *to*. For example:

```
SELECT TRANSLATE('foo', 'fo', 'ab') -> 'abb'
```

TRUNC(float8)

Truncates (towards zero). For example:

```
SELECT TRUNC( PI() ) -> 3
```

VARCHAR(string)

Converts *string* to a *VARCHAR*.

WIDTH(box)

Returns the width of *box*. For example:

```
SELECT WIDTH( BOX '((0,0),(3,1))' ) -> 3
```

4.5.5 SQL Server-Supported Functions

This section provides an alphabetical listing of Microsoft SQL Server-supported functions, with examples and corresponding results.

ACOS(number)

Returns the arc cosine of *number* ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

APP_NAME()

Returns the application name for the current session, set by the application. For example:

```
SELECT APP_NAME( ) -> 'SQL Enterprise Manager'
```

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(number)

Returns the arc sine of *number* ranging from -1 to 1. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(number)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATN2(float1, float2)

Returns the angle (in radians) whose tangent is *float1/float2*. For example:

```
SELECT ATN2( 35.175643, 129.44 ) -> 0.265345
```

*BINARY_CHECKSUM(* / expression [,...n])*

Returns the binary checksum for a list of expressions or for a row of a table. This example returns a list of user IDs where the stored password checksum doesn't match the current password's checksum:

```
SELECT userid AS 'Changed' FROM users WHERE NOT password_chksum =  
BINARY_CHECKSUM( password )
```

CHAR(integer_expression)

Converts a numeric ASCII code to a character. For example:

SELECT CHAR(78) -> 'N'

CHARINDEX(substring, string [, start_location])

Returns the position of the first occurrence of a *substring* in a *string*. For example:

SELECT CHARINDEX('he', 'Howdy, there!') -> 9

*CHECKSUM(* / expression [,...n]*)

Returns a checksum (computed over row values or expressions provided). The following example returns a list of user IDs for which the stored password checksum doesn't match the current password's checksum:

**SELECT userid AS 'Changed' FROM users WHERE NOT password_chksum =
BINARY_CHECKSUM(password)**

CHECKSUM_AGG([ALL | DISTINCT] expression)

Returns the checksum of the values in a group. For example:

SELECT CHECKSUM_AGG(BINARY_CHECKSUM(*)) FROM authors -> 67

COALESCE(expression [,...n])

Returns the first non-NULL argument from a list of arguments. For example:

SELECT COALESCE(NULL, 1, 3, 5, 7) -> 1

COL_LENGTH(table, column)

Returns *column* length in bytes. For example:

SELECT COL_LENGTH('authors', 'au_fname') -> 50

COL_NAME(table_id, column_id)

Returns column name, given *table_id* and *column_id*. For example:

SELECT COL_NAME(OBJECT_ID('authors'), 1)

*CONTAINS({column | *}, contains_search_condition)*

Searches *columns* for exact or "fuzzy" matches of the *contains_seach_criteria*. *CONTAINS* is an elaborate function used to perform full-text searches. Refer to the vendor documentation for more information. This example returns all product ID's from the **products** table that contain the words "peanut" and "butter" in close proximity to each other.

**SELECT productid FROM products
WHERE CONTAINS(productname, ' "peanut" NEAR "butter" ')**

CONTAINSTABLE(table, column, contains_search_condition)

Returns a *table* with exact and "fuzzy" matches to *contains_search_condition*. *CONTAINSTABLE* is an elaborate function used to perform full-text searches. The following example returns all product ID's from the **products** table that contain the words "peanut" and "butter" in close proximity to each other:

**SELECT productid FROM products WHERE CONTAINS
(products, productname, ' "peanut" NEAR "butter" ')**

CONVERT(data_type[(length)], expression [, style])

Converts data from one datatype to another. For example:

SELECT CONVERT(VARCHAR(50), CURRENT_TIMESTAMP, 1) -> '06/29/03'

COS(number)

Returns the cosine of *number* as an angle expressed in radians. For example:

SELECT COS(0) -> 1.000000

COT(number)

Returns the cotangent of *number*. For example:

SELECT COT(3.1415) -> -10792.88993953

DATABASEPROPERTYEX (database, property)

Returns a database option or property. For example:

SELECT DATABASEPROPERTYEX('pubs', 'Version') -> 539

DATALENGTH(expression)

Returns the number of bytes in a character or binary string. For example:

SELECT MAX(DATALENGTH(au_fname)) FROM authors -> 11

DATEADD(datepart, number, date)

Adds a number of *dateparts* (e.g., days) to a datetime value. For example:

SELECT DATEADD(Year, 10, CURRENT_TIMESTAMP) -> 2013-06-29 19:47:15.270

DATEDIFF(datepart, startdate, enddate)

Calculates the difference between two datetime values expressed in the specified *datepart*. For example:

**SELECT DATEDIFF(Day, CURRENT_TIMESTAMP,
DATEADD(Year, 1, CURRENT_TIMESTAMP))**

366

DATENAME(datepart, date)

Returns the name of a *datepart* (e.g., month) of a datetime argument. For example:

SELECT DATENAME(month, GETDATE()) -> 'June'

DATEPART(datepart, date)

Returns the value of a *datepart* (e.g., hour) of a datetime argument. For example:

SELECT DATEPART(year, GETDATE()) -> 2003

DAY(date)

Returns an integer value representing the day of the *date* provided as a parameter. For example:

SELECT DAY('04/15/2004') -> 15

DB_ID([database_name])

Returns a database ID and given name. For example:

SELECT DB_ID() -> 5

DB_NAME(database_id)

Returns the database name. For example:

SELECT DB_NAME(5) -> 'pubs'

DEGREES(numeric_expression)

Converts radians to degrees. For example:

SELECT DEGREES(PI()) -> 180

DIFFERENCE(character_expression, character_expression)

Compares how two arguments sound and returns a number from 0 to 4. A higher result indicates a better phonetic match. For example:

SELECT DIFFERENCE('moe', 'low') -> 3

FILE_ID(file_name)

Returns the file ID for the logical *file_name*. For example:

SELECT FILE_ID('master') -> 1

FILE_NAME(file_id)

Returns the logical filename for the *file_id*. For example:

SELECT FILE_NAME(1) -> 'master'

FILEGROUP_ID(filegroup_name)

Returns filegroup ID for the logical *filegroup_name*. For example:

SELECT FILEGROUP_ID('PRIMARY') -> 1

FILEGROUP_NAME(filegroup_id)

Returns the logical filegroup name for *filegroup_id*. For example:

```
SELECT FILEGROUP_NAME( 1 ) -> 'PRIMARY'
```

FILEGROUPPROPERTY(filegroup_name, property)

Returns the filegroup property value for the specified *property*. For example:

```
SELECT FILEGROUPPROPERTY( 'PRIMARY', 'IsReadOnly' ) -> 0
```

FILEPROPERTY(file, property)

Returns the *file* property value for the specified *property*. For example:

```
SELECT FILEPROPERTY( 'pubs', 'SpaceUsed' ) -> 160
```

FULLTEXTCATALOGPROPERTY(catalog_name, property)

Returns the fulltext catalog properties. For example:

```
SELECT FULLTEXTCATALOGPROPERTY( 'Cat_Desc', 'LogSize' )
```

FULLTEXTSERVICEPROPERTY(property)

Returns the fulltext service level properties. For example:

```
SELECT FULLTEXTSERVICEPROPERTY('IsFulltextInstalled') -> 1
```

FORMATMESSAGE(msg_number, param_value [,... n])

Constructs a message from an existing message in the *SYSMESSAGES* table. (Similar to *RAISEERROR*). For example:

```
sp_addmessage 50001, 1, 'Table %s has %s rows.'
```

```
SELECT FORMATMESSAGE(50001, 'AUTHORS',
```

```
(SELECT COUNT(*) FROM AUTHORS) ) 'Table AUTHORS has 23.'
```

FREETEXT({ column |}, freetext_string)*

Used for a full-text search. Returns rows with *column* that match the meaning, but not exactly the value, of *freetext_string*.

FREETEXTTABLE(table, { column |}, freetext_string [, top_n_by_rank])*

Used for a full-text search. Returns rows from *table* with *column* that match the meaning, but not exactly the value, of *freetext_string*. For example:

```
SELECT * from FREETEXTTABLE (authors, *, 'kev')
```

GETANSINULL([database])

Returns default nullability setting for new columns. For example:

```
SELECT GETANSINULL( ) -> 1
```

GETDATE()

Returns current date and time. For example:

```
SELECT GETDATE( ) -> 2003-06-27 19:26:59.893
```

GETUTCDATE()

Returns the current date as a Coordinated Universal Time (UTC) date. For example:

```
SELECT GETUTCDATE( ) -> 2003-06-28 02:26:46.720
```

GROUPING(column_name)

Returns 1 when a row is added by *CUBE* or *ROLLUP*; otherwise returns 0. For example:

```
SELECT royalty, SUM(advance) 'total advance',  
       GROUPING(royalty) 'grp'  
FROM titles  
GROUP BY royalty WITH ROLLUP
```

royalty	total advance	grp
-----	-----	---
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

HOST_ID()

Returns the workstation ID. For example:

```
SELECT HOST_ID( ) -> 216
```

HOST_NAME()

Returns the process host name. For example:

```
SELECT HOST_NAME( ) -> 'PLATO'
```

IDENT_CURRENT(table_name)

Returns the last identity value generated for the specified table. For example:

```
SELECT IDENT_CURRENT('jobs') -> 876
```

IDENT_INCR(table_or_view)

Returns an identity column increment value. For example:

```
SELECT IDENT_INCR('jobs') -> 1
```

IDENT_SEED(table_or_view)

Returns an identity seed value. For example:

```
SELECT IDENT_SEED('jobs') -> 1
```

IDENTITY(data_type [, seed, increment]) AS column_name

Used in a *SELECT INTO* statement to insert an identity column into the destination table. For example:

```
SELECT IDENTITY(int, 1,1) AS ID
```

```
INTO NewTable
```

```
FROM OldTable
```

INDEX_COL(table, index_id, key_id)

Returns an index column name given a table name, index ID, and the sequential number of the column in the index key. For example:

```
SELECT INDEX_COL(OBJECT_ID('authors'), 1, 1) -> NULL
```

INDEXPROPERTY(table_id, index, property)

Returns an index property (such as *FILLFACTOR*). For example:

```
SELECT INDEXPROPERTY(OBJECT_ID('authors'),
```

```
'UPKCL_auind', 'IsPadIndex')
```

```
0
```

ISDATE(expression)

Validates if a character string can be converted to *DATETIME*. For example:

```
SELECT ISDATE(NULL), ISDATE(GETDATE( ))
```

```
0 1
```

IS_MEMBER({group | role})

Returns true or false (1 or 0) depending on whether the user is a member of the specified Windows NT *group* or SQL Server *role*. For example:

```
SELECT IS_MEMBER( 'db_owner' ) -> 0
```

IS_SRVROLEMEMBER(role [,login])

Returns true or false (1 or 0) depending on whether the user is a member of the specified server *role*. For example:

```
SELECT IS_SRVROLEMEMBER( 'sysadmin' ) -> 0
```

ISNULL(check_expression, replacement_value)

Returns the first argument if it is not NULL; otherwise returns the second argument. For example:

SELECT ISNULL(NULL, 'NULL') -> 'NULL'

ISNUMERIC(expression)

Validates if a character string can be converted to *NUMERIC*. For example:

SELECT ISNUMERIC('3.1415'), ISNUMERIC('IRK')

1 0

LEFT(character_expression, integer_expression)

Returns the leftmost *integer_expression* characters of *character_expression*. For example:

SELECT LEFT('Wet Paint', 3) -> 'Wet'

LEN(string_expression)

Returns the number of characters in the expression. For example:

SELECT LEN('Wet Paint') -> 9

LOG(float_expression)

Returns the natural logarithm. For example:

SELECT LOG(PI()) -> 1.1447298858494002

LOG10(float_expression)

Returns the base-10 logarithm. For example:

SELECT LOG10(PI()) -> 0.49714987269413385

LTRIM(character_expression)

Trims leading space characters. For example:

SELECT LTRIM(' beaucoup ') -> 'beaucoup '

MONTH(date)

Returns the month part of the date provided. For example:

SELECT MONTH(GETDATE()) -> 6

NCHAR(integer_expression)

Returns the UNICODE character with the given integer code. For example:

SELECT NCHAR(120) -> 'x'

NEWID()

Creates a new unique identifier of type *UNIQUEIDENTIFIER*. For example:

SELECT NEWID() -> '32B35185-F55E-4FE0-B2C8-B57B35815C12'

NULLIF(expression, expression)

Returns a NULL if the two specified expressions are equivalent. For example:

SELECT NULLIF(5, 5) -> NULL

OBJECT_ID(object)

Returns the object ID of *object*. For example:

SELECT OBJECT_NAME (OBJECT_ID('authors')) -> 'authors'

OBJECT_NAME(object_id)

Returns the object name of an object with the object ID. For example:

SELECT OBJECT_NAME (OBJECT_ID('authors')) -> 'authors'

OBJECTPROPERTY(id, property)

Returns the properties of objects in the current database. For example:

SELECT OBJECTPROPERTY (object_id('authors'),'ISTABLE') -> 1

OPEN {[GLOBAL]cursor_name}| cursor_variable_name}

Opens a local or global cursor.

OPENDATASOURCE(provider_name, init_string)

Makes a connection to a data source without using a linked server name. For examples, refer to the "Loaders" section of the SQL Server User's Guide.

OPENQUERY(linked_server, query)

Queries a remote data source previously configured as a linked server. For an example, refer to the "Loaders" section of the SQL Server User's Guide.

OPENROWSET(provider_name, {datasource; user_id,password | provider_string}, {[catalog.][schema.]object | query})

Queries a remote data source without setting it up as a linked server.

PARSENAME(object_name, object_piece)

Returns the database name, owner name, server name, or object name for the object specified. *Object_piece* is an integer between 1 and 4. For example:

SELECT PARSENAME('pubs..authors', 1) -> 'authors'

SELECT PARSENAME('pubs..authors', 2) -> NULL

SELECT PARSENAME('pubs..authors', 3) -> 'pubs'

SELECT PARSENAME('pubs..authors', 4) -> NULL

PATINDEX('pattern%', expression)

Returns the position of the first occurrence of a pattern in a string. For example:

SELECT PATINDEX('%Du%', 'Donald Duck') -> 8

PERMISSIONS([object_id [, column]])

Returns a numeric value representing a bitmap with the current user's permissions on the specified object or column. For example:

SELECT PERMISSIONS(OBJECT_ID('authors'))&8 -> 8

PI()

Returns the pi constant. For example:

SELECT 2*PI() -> 6.2831853071795862

RADIANS(numeric_expression)

Converts degrees to radians. For example:

SELECT RADIANS(90.0) -> 1.570796326794896600

RAND([seed])

Returns a pseudo-random *FLOAT* type value between and 1. For example:

SELECT RAND(PI()) -> 0.71362925915543995

REPLICATE(character_expression, integer_expression)

Repeats a string a number of times. For example:

SELECT REPLICATE('FOOBAR', 3) -> 'FOOBARFOOBARFOOBAR'

REPLACE(string_expression1, string_expression2, string_expression3)

Performs a search-and-replace on *string_expression1*, replacing each occurrence of *string_expression2* with *string_expression3*. For example:

SELECT REPLACE('Donald Duck', 'Duck', 'Trump') -> 'Donald Trump'

REVERSE(character_expression)

Reverses the characters of a string. For example:

SELECT REVERSE('Donald Duck') -> 'kcuD dlanoD'

RIGHT(character_expression, integer_expression)

Returns the rightmost *integer_expression* characters of *character_expression*. For example:

SELECT RIGHT('Donald Duck', 4) -> 'Duck'

ROUND (number, decimal [,function])

Returns *number* rounded to *decimal* places right of the decimal point. Note that *decimal*, an integer, can be negative to round off digits left of the decimal point. If a nonzero integer is provided for *function*, the return value will be truncated; otherwise the value is rounded. For example:

```
SELECT ROUND(PI( ), 2) -> 3.1400000000000001
```

ROWCOUNT_BIG()

Returns the number of rows affected by the most recent query. (Same as @@*ROWCOUNT*, but returns a *BIGINT* type.) For example:

```
SELECT ROWCOUNT_BIG( ) -> 1
```

RTRIM(character_expression)

Trims trailing space characters from the expression. For example:

```
SELECT RTRIM(' beaucoup ') -> ' beaucoup'
```

SIGN(numeric_expression)

Returns -1 if the argument is negative, 0 if it is zero, and 1 if the argument is positive. For example:

```
SELECT SIGN(-PI( )) -> -1.0
```

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

SOUNDEX(character_expression)

Returns a four-character code based on how the argument string sounds. For example:

```
SELECT SOUNDEX('char') -> 'C600'
```

SPACE(integer_expression)

Returns a string consisting of a given number of space characters. For example:

```
SELECT SPACE(5) -> '     '
```

STATS_DATE(table_id, index_id)

Returns the date and time that index statistics were last updated. For example:

```
SELECT i.name, STATS_DATE(i.id, i.indid)
```

```
FROM sysobjects o, sysindexes i
```

```
WHERE o.name = 'authors' AND o.id = i.id
```

```
UPKCL_aidind 2000-08-06 01:34:00.153
```

```
 aunmind    2000-08-06 01:34:00.170
```

STDEV(expression)

Returns the standard deviation of values in *expression*. For example:

```
SELECT STDEV( qty ) FROM sales -> 16.409201831957116
```

STDEVP(expression)

Returns the standard deviation for the population of values in *expression*. For example:

```
SELECT STDEVP( qty ) FROM sales -> 16.013741264834152
```

STR(number [, length [, decimal]])

Converts *number* to a character string with *length* length and *decimal* decimal places.

STUFF(string1, start, length, string2)

Replaces the *length* characters at position *start* within *string1* with those in *string2*. For example:

```
SELECT STUFF( 'Donald Duck', 8, 4, 'Trump' ) -> 'Donald Trump'
```

SUBSTRING(string, start, length)

Extracts *length* characters from *string* starting at the character in the *start* position. For example:

```
SELECT SUBSTRING( 'Donald Duck', 8, 4 ) -> 'Duck'
```

SUSER_ID([login])

Returns the system user ID of a given login name. Incidentally, this function will always return NULL with SQL Server 2000 or later. Therefore, avoid using this function.

SUSER_SID([login])

Returns the Security ID (SID) for the current user, or for the specified *login*. The SID is returned in binary format. For example:

```
SELECT SUSER_SID('montoyai')
```

```
0x68FC17A71010DE40B005BCF2E443B377
```

SUSER_SNAME([server_user_sid])

Returns the login name for the current user, or for the specified login Security ID (SID). For example:

```
SELECT SUSER_SNAME( ) -> 'montoyai'
```

TAN(number)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TEXTPTR(column)

Returns a pointer to a *TEXT*, *NTEXT*, or *IMAGE* column in *VARBINARY* format. For example:

```
SELECT TEXTPTR(pr_info)
```

```
FROM pub_info WHERE pub_id = '0736'  
  
ORDER BY pub_id  
  
0xFEFF6F0000000005C00000001000100
```

TEXTVALID(table.column, text_ptr)

Returns true or false (1 or 0), depending on whether the provided pointer to a *TEXT*, *NTEXT*, or *IMAGE* column is valid. For example:

```
SELECT pub_id, 'Valid (if 1) Text data'  
      = TEXTVALID ('pub_info.logo', TEXTPTR(logo))  
  
FROM pub_info  
  
ORDER BY pub_id  
  
0736    1  
0877    1  
1389    1  
1622    1  
1756    1  
9901    1  
9952    1  
9999    1
```

TYPEPROPERTY(datatype, property)

Returns information about datatype properties. The *datatype* argument can contain the name of any datatype, and *property* can be a string containing one of the following:

Precision

The precision of the *datatype* is the number of digits or characters that it can store.

Scale

The scale is the number of decimal places for a numeric datatype. A NULL value will be returned if *datatype* is not a numeric datatype. For example:

```
SELECT TYPEPROPERTY('decimal', 'PRECISION') -> 38
```

UNICODE(ncharacter_expression)

Returns the UNICODE code point for the first character of the input parameter. For example:

```
SELECT UNICODE('Hello!') -> 72
```

USER_ID([user])

Returns the user ID for *user* in the current database. If *user* is omitted, then the current user's ID will be returned. For example:

```
SELECT USER_ID( ) -> 2
```

USER_NAME([id])

Returns the current username in the current database. For example:

SELECT USER_NAME() -> 'montoyai'

VAR(expression)

Returns the statistical variance for the values represented by *expression*. For example:

SELECT VAR(qty) FROM sales -> 269.26190476190476

VARP(expression)

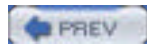
Returns statistical variance for the population represented by all values of *expression* in a group. *VARP* is an aggregate function. For example:

SELECT VARP(qty) FROM sales -> 256.43990929705217

YEAR(date)

Returns an integer that is the *YEAR* part of the specified date. For example:

SELECT YEAR(CURRENT_TIMESTAMP) -> 2003



Chapter 4. SQL Functions

A function is a special type of command word in the SQL command set, and each SQL dialect varies in its implementation of the command set. In effect, functions are one-word commands that return single values. The value of a function can be determined by input parameters, as with a function that averages a list of database values. But many functions do not use any type of input parameter, such as the function that returns the current system time, *CURRENT_TIME*.

The ANSI standard supports a number of useful functions. This chapter covers those functions, providing detailed descriptions and examples for each platform. In addition, each database maintains a long list of their own internal functions that are outside of the scope of the standard SQL. This chapter provides parameters and descriptions for each database implementation's internal functions.

In addition, most database platforms support the ability to create user-defined functions (UDFs). For more information on UDFs, refer to the [CREATE/ALTER FUNCTION/PROCEDURE Statements](#).

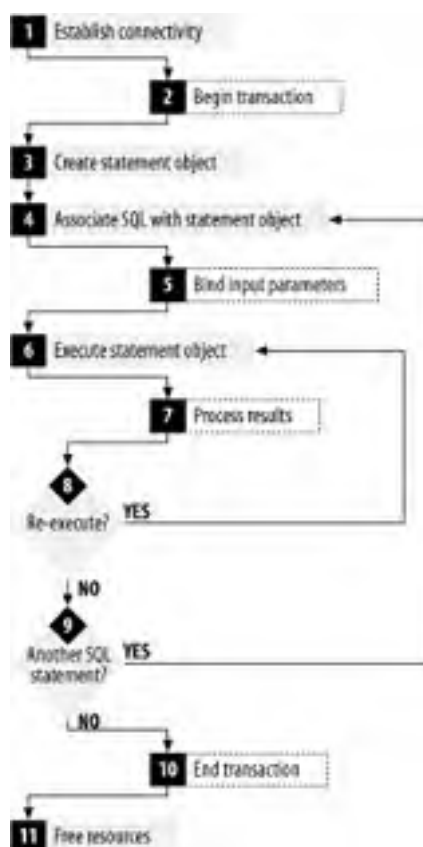
5.1 Database Programming Overview

Developing successful database applications, large and small, involves many steps. Careful thought must be given to application architecture, and especially to the following issues:

- How to map application data, which is typically object-oriented, to a relational database
- How to handle errors gracefully
- How to maximize performance and scalability

A typical database application will require many different SQL statements. The management of so many statements is simplified by the fact that all SQL statements will follow roughly the same pattern of execution within an application. [Figure 5-1](#) is a state diagram showing how SQL statements are prepared, executed, and then processed by a database application when interacting with a relational database system. The state diagram has been broken down into eleven steps, four of which are optional (and are indented in the diagram).

Figure 5-1. Statement execution state diagram



Following are detailed descriptions of each step shown in [Figure 5-1](#):

1. *Establish connectivity*: Establishing connectivity is the first step in every successful database application. It is in this step that the client, or database application, makes a physical connection to the database that will be used to transmit the SQL statements to the database and the results back to the client. The actual physical connection could be over a LAN, WAN, or even a simple logical connection back to the client in cases where the database application and server are running on the same machine. For more information on how to establish connectivity, please see the section [Section 5.2](#) later in this chapter.
2. *Begin transaction (optional)*: A transaction may be begun so that the database changes may be rolled back on failure or committed on success. For more information on transaction control from database programming APIs, please see [Section 5.4](#) later in this chapter.

3. *Create statement object*: Most modern database programming APIs are object-oriented and therefore use an object to represent a SQL statement. There will typically be one statement object per SQL statement executed within the application. The statement object holds the state information required to execute the SQL statement, such as the SQL statement itself and input parameters when they exist.
4. *Associate SQL with statement object*: After the statement object has been created, it needs to have a SQL statement assigned to it. Once this is done, the statement object may be executed.
5. *Bind input parameters (optional)*: While not part of the ANSI SQL standard, the ability to bind parameters to "placeholders" within a SQL statement is supported by all the database platforms covered in this book. If the SQL statement has placeholders for input parameters, the statement object will need to have a program variable associated with each input parameter. If the SQL statement contains no input parameters, then this step may be skipped. Input parameters are useful for optimizing performance when the same SQL statement is re-executed many times, because the server-side parsing of the statement only needs to be done once prior to the first execution. Another reason to use input parameters is to embed binary data, such as *BLOB* data, into SQL statements such as *INSERT* and *UPDATE* statements.
6. *Execute statement object*: After the statement object has been successfully created and initialized with a SQL statement, the statement object can be executed. This step executes the SQL statement on the database server.
7. *Process Results (optional)*: After the database server returns a result set, the application may process the results. This step is optional, since it is typically not required for statements that insert or update data in the database.
8. *Re-execution*: If the same statement needs to be re-executed to retry on a failure or to execute again with different values for the input parameters, the application can return to Step 6. If the application has no need to re-execute the same statement, it moves on to Step 9.
9. *Execute another SQL statement*: If the application needs to execute a different SQL statement and can reuse the statement object, then the application can return to Step 4; if not, it can move to Step 10.
10. *End transaction (optional)*: Assuming a transaction was begun in step 2, the transaction is now either committed or rolled back. If the transaction is rolled back, then all changes to the database made by the statement object will be removed from the database.
11. *Free resources*: After successfully executing the statement and processing any results, the client- and server-side resources need to be freed for use by other applications.

The remaining sections in this chapter provide examples of how to use the ADO.NET and JDBC database APIs to build applications that follow the steps outlined in [Figure 5-1](#).

5.2 Opening a Database Connection

Before interacting with a database, an application must first establish a connection to the database server. The APIs covered in this book abstract the low-level connectivity details into a few simple object-oriented classes, making it easier to focus on the database application instead of protocols and network topology.

5.2.1 Opening an ADO.NET Database Connection

Opening a connection with ADO.NET requires instantiating a connection object with a properly formatted connection string and then invoking the `Open` method on the connection object. The connection object can be an `OdbcConnection`, `SqlConnection`, or an `OleDbConnection`. The `OdbcConnection` is designed for any ODBC datasource, and the `OleDbConnection` type will work with any OLE DB Provider. For the highest performance data access, use connection objects specifically tuned to the specific database platform, such as `SqlConnection` for Microsoft SQL Server. Following is the syntax for creating a `Connection` object in ADO.NET:

```
{Odbc|OleDb|Sql}Connection connection =  
    new {Odbc|OleDb|Sql}Connection(connection_string);  
connection.Open( );
```

The format of the connection string is the same for all of the connection types. The format is a string of key/value pairs delimited by semicolons. For example:

```
key1=value1; key2=value2; key3=value3; ...
```

While the format is the same for every connection type, the keys and values are quite different. Tables [Table 5-1](#) through [Table 5-3](#) list the attributes for the three connection types listed above. Many database platforms support additional attributes that can also be set through the connection string. For a list of these attributes, please consult the appropriate database vendor documentation.

Following are examples of two connection strings for an `OdbcConnection`:

```
DSN=MyOracleDSN; UID=scott; PWD=tiger;  
DRIVER={SQL Server};SERVER=(local);UID=sa;PWD=;DATABASE=pubs;
```

The first string connects to a Data Source Name, or DSN, with the name **MyOracleDSN** using the username **scott** and the password **tiger**.

The second string connects using the **SQL Server** driver to a database named **pubs** on the local server. The username is **sa**. The password is blank, indicating to the driver that the password is not required for the **sa** user.

Following is an example of a connection string for an `OleDbConnection`, which connects to an Oracle9i data source using the MSDAORA OLE DB provider, the use name **scott**, and the password **tiger**.

```
Provider=MSDAORA;Data Source=Oracle9;User ID=scott;Password=tiger;
```

Finally, here is an example of a connection string for a `SqlConnection`, which connects to a SQL Server data source on the local server:

```
Server=(local);UID=sa;PWD=;DATABASE=pubs;Connection Timeout=60;
```

In Tables [Table 5-1](#) through [Table 5-3](#), you'll find synonyms for some keywords. For example, you can use "DSN" and "Data Source Name" interchangeably in your code.

Table 5-1. Connection string attributes for OdbcConnection

Keyword	Notes
---------	-------

Data Source Name DSN	The DSN, FILEDSN, or DRIVER attribute must be provided to connect. The DSN is the Data Source Name known by the client's driver manager. The advantage of using a DSN is that the DSN can be changed to point to a different database platform while the database applications are still running.
User ID UID	This value is set to the user identifier authorized to open a new connection.
Password PWD	The password for the user. If the user doesn't have a password for the data source, then this property should still be supplied with an empty value.
DSN_File Name FILEDSN	Similar to the DSN, the FILEDSN is a file, typically with a DSN extension that contains the attributes for the connection object to establish connectivity. Even when using a FILEDSN, the connection may still need the password provided, since passwords aren't stored within DSN files.
Driver Name DRIVER	A Driver may be explicitly used instead of a DSN or FILEDSN. The drawback of using a driver directly is that the application won't be isolated from changes made to the driver and will require slight modifications to target a different database platform.
SAVEFILE	When the SAVEFILE attribute is set to a legal filename, the connection attributes will be saved to the file once successful connectivity is established. This option is only available when using DRIVER and FILEDSN connectivity methods.

Table 5-2. Connection string attribute for OleDbConnection

Keyword	Notes
Provider	The name of the provider: this is the only required attribute. The specified provider may require additional attributes.

Table 5-3. Connection string attributes for SqlConnection

Keyword	Notes
Application Name	The name of the application that will be displayed from server management utilities.
AttachDBFilename Extended properties Initial File Name	The pathname to an attachable database.
Connect Timeout Connection Timeout	Number of seconds to wait before the connection attempt is aborted. The default is 15 seconds.
Connection Lifetime	Time in seconds the connection can remain in the connection pool, when pooling is enabled.
Connection Reset	Specifies whether or not the connection is reset when it is reused from the connection pool. Default is true.
Current Language	The language the connection session should use.
Data Source Server Address Addr Network Address	The SQL Server instance name or network address.
Initial Catalog Database	The name of the database.
Integrated Security Trusted_Connection	Set to true or <code>sspi</code> for secure connections. The default is false.

Max Pool Size	The maximum number of connections allowed in the connection pool at a time. The default is 100.
Min Pool Size	The minimum number of connections to keep in the connection pool at a time. The default is 0.
Network Library Net	The network library to use in establishing the connection. Valid settings are <code>dbnmpntw</code> for Named Pipes, <code>dbmsrpcn</code> for Multiprotocol, <code>dbmsadsn</code> for AppleTalk, <code>dbmsgnet</code> for VIA, <code>dbmsipcn</code> for Shared Memory, <code>dbmsspxn</code> for IPX/SPX, and <code>dbmssocn</code> for TCP/IP. The default is <code>dbmssocn</code> for TCP/IP.
Packet Size	The network packet size in bytes; the default is 8,192.
Password Pwd	The user's password.
Persist Security Info	Determines if security-sensitive connection properties, such as the password, are stored within the connection object after a connection has been attempted or completed. The default is false.
Pooling	Determines if connection pooling should be used for the connection. The default is true.
User ID	The user's login name.
Workstation ID	The name of the computer connecting to the database.

5.2.2 Opening a JDBC Database Connection

Following is the syntax for registering a driver with the JDBC Driver Manager and then opening a database connection:

```
Class.forName(driver_name);  
  
Connection connection = DriverManager.getConnection(connect_string,  
                                                username, password);
```

The first step in establishing JDBC connectivity is to instruct the Java Virtual Machine (JVM) class loader to load the appropriate JDBC driver.

The most common method of loading the driver into the class loader is to use the static `forName` method of the `Class` class. This method can provide applications with greater flexibility in changing database platforms by having the Java Virtual Machine (JVM) load the database driver at runtime:

```
Class.forName(" driver_name ");
```

After the database driver has been loaded, the application can establish connectivity by invoking the static `getConnection` method on the JDBC `DriverManager` class. The `getConnection` method takes three arguments: a connection string, username, and password:

```
Connection connection = DriverManager.getConnection(connect_string,  
                                                username,      password);
```

The connection string follows the following JDBC URL naming scheme:

```
jdbc:subprotocol.subname
```

Following are examples of how to connect to the different vendors covered in this book using JDBC.

5.2.2.1 DB2

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");  
  
Connection connection = DriverManager.getConnection(  
    "jdbc:db2:DATABASE", "user", "passwd" );
```

5.2.2.2 MySQL

```
Class.forName("org.gjt.mm.mysql.Driver");  
Connection connection = DriverManager.getConnection(  
    "jdbc:mysql://127.0.0.1:3306/DATABASE",  
    "user", "passwd" );
```

5.2.2.3 PostgreSQL

```
Class.forName("org.postgresql.Driver");  
Connection connection = DriverManager.getConnection(  
    "jdbc:postgresql://127.0.0.1:5432/DATABASE",  
    "user", "passwd" );
```

5.2.2.4 Oracle

```
Class.forName("oracle.jdbc.driver.OracleDriver");  
Connection connection = DriverManager.getConnection(  
    "jdbc:oracle:thin:@myserver",  
    "scott", "tiger" );
```

5.2.2.5 SQL Server

```
Class.forName(  
    "com.microsoft.jdbc.sqlserver.SQLServerDriver"  
);  
Connection connection = DriverManager.getConnection(  
    "jdbc:microsoft:sqlserver://SERVER:1433;" +  
    "DatabaseName=pubs;",  
    "user", "passwd" );
```

5.3 Closing a Database Connection

Once an application is finished using an open connection, that connection should be closed. Some APIs automatically close connections when the application exits; however, it is best to explicitly close the connection as soon as the application is finished accessing the database to allow other applications access to precious connection resources.



Calling the appropriate close methods when finished with an ADO.NET or JDBC object is essential to freeing precious database resources used by the application. Both ADO.NET and Java offer very few guarantees when objects are finalized; therefore, it is common for those applications that don't close objects explicitly to run out of database resources (such as connections, statement handles, and memory).

5.3.1 Closing an ADO.NET Database Connection

To close an *ADO.NET* connection, invoke the `Close` method on the connection object:

```
connection.Close( );
```

5.3.2 Closing a JDBC Database Connection

To close a *JDBC* connection, invoke the `close` method on the connection object:

```
connection.close( );
```

5.4 Managing Transactions

Most database programming APIs provide methods of controlling transactions, setting savepoints, and modifying isolation levels. This section covers the mechanisms for controlling transactions from the ADO.NET and JDBC APIs covered in this chapter.

5.4.1 Beginning a Transaction

Beginning a transaction is the first step in guaranteeing atomicity when executing multiple SQL statements. After beginning a transaction, the transaction can be committed to make the changes performed by the executed SQL statements permanent or the transaction can be rolled back to abort the changes and leave the database unchanged.

5.4.1.1 Beginning an ADO.NET transaction

To begin a transaction using ADO.NET, invoke the `BeginTransaction` method on a connection object. A `Transaction` object is returned that can be used in the creation of ADO.NET `Command` objects that execute within the same transaction. To execute a `Command` object within the new transaction, you must first attach the `Transaction` object to the `Command` object's `Transaction` property. Following is the syntax for starting a transaction by creating a `Transaction` object in ADO.NET:

```
{Odbc|OleDb|Sql}Transaction transaction =  
    connection.BeginTransaction([IsolationLevel.  
        {Chaos | ReadCommitted | ReadUncommitted | RepeatableRead |  
        Serializable | Unspecified}]);  
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
statement.Transaction = transaction;
```

When creating the `Transaction` object, you can optionally specify the *isolation level* that is used for the transaction. The isolation level controls how much (or little) your database transactions are insulated from the effects of other transactions. The available isolation levels are:

Chaos

Pending changes of other transactions with higher isolation levels (`ReadCommitted`, `ReadUncommitted`, `RepeatableRead`, or `Serializable`) cannot be changed.

ReadUncommitted

The lowest ANSI standard isolation level, unprotected against dirty reads, non-repeatable reads, and phantom records.

ReadCommitted

The next highest ANSI standard isolation level above `ReadUncommitted`, protects against dirty reads.

RepeatableRead

The next highest ANSI standard isolation level above `ReadCommitted`, protects against dirty reads and nonrepeatable reads.

Serializable

The highest ANSI standard isolation level. Protects against dirty reads, non-repeatable reads, and phantom records.

Unspecified

Isolation level cannot be determined. This option is not typically passed into the `BeginTransaction` method; its purpose is to provide an Unknown state for the `Connection` object's `IsolationLevel` property. If the `Transaction` has this `IsolationLevel`, even after being set to another value, then it is safe to assume that the database does not support the `IsolationLevel` that you requested.

For more information about isolation levels, see [Programming Tips and Gotchas](#) under [SET TRANSACTION Statement](#).



The isolation level used during SQL statement execution can have a significant impact on the performance and scalability of a database application, as well as on the RDBMS itself. When deciding which isolation level to use, choose the lowest level that provides the appropriate isolation guarantees for the database application.

5.4.1.2 Beginning a JDBC transaction

JDBC starts a connection in `AUTO COMMIT` mode, which is a setting that automatically commits changes made by each SQL statement when it executes. To gain control over when changes are committed, the `AUTO COMMIT` setting must be turned off. JDBC's mechanism of beginning a transaction is subtle compared to ADO.NET, which uses a `BeginTransaction` method to begin a transaction. In JDBC, a transaction is started as soon as the `AUTO COMMIT` mode is turned off or just after the transaction has been committed or rolled back. Committing or rolling back the transaction doesn't turn the `AUTO COMMIT` mode back on, so multiple commits or rollbacks will need to be executed to commit/roll back multiple units of work until the connection is closed or `AUTO COMMIT` is explicitly turned back on. To turn off the `AUTO COMMIT` mode, execute the `setAutoCommit` method of the JDBC `Connection` object with a false argument:

```
connection.setAutoCommit( false );
```



Beware that invoking `setAutoCommit` to turn `AUTO COMMIT` mode on or off when a transaction has already begun will automatically commit that transaction.

JDBC also supports different isolation levels through the JDBC `Connection` interface. To set the isolation level used for new transactions, invoke the `setTransactionIsolation` method with one of the following isolation levels:

TRANSACTION_NONE

Indicates transactions are not supported on the connection. This value is not typically used to set the isolation level, but to query the connection for transaction support with the `getTransactionIsolation` method on the `Connection` interface.

TRANSACTION_READ_UNCOMMITTED

The lowest ANSI standard isolation level, unprotected against dirty reads, nonrepeatable reads, and phantom records.

TRANSACTION_READ_COMMITTED

The next highest ANSI standard isolation level above `TRANSACTION_READ_UNCOMMITTED`. Protects against dirty reads.

TRANSACTION_REPEATABLE_READ

The next highest ANSI standard isolation level above `TRANSACTION_READ_COMMITTED`. Protects against dirty reads and nonrepeatable reads.

TRANSACTION_SERIALIZABLE

The highest ANSI standard isolation level. Protects against dirty reads, nonrepeatable reads, and phantom records.

For a more detailed explanation of transaction isolation levels, see [Programming Tips and Gotchas](#) under [SET TRANSACTION Statement](#).

Following is example code setting the isolation level in JDBC:

```
connection.setIsolationLevel( Connection.TRANSACTION_SERIALIZABLE );
```



The isolation level used on the connection can have a significant impact on the performance and scalability of a database application, as well as on the RDBMS itself. When deciding which isolation level to use, choose the lowest level that provides the appropriate isolation guarantees for the connection.

5.4.2 Committing a Transaction

Committing a transaction is a way to explicitly close the transaction and make its database modifications permanent.

5.4.2.1 Committing an ADO.NET transaction

To commit a transaction using ADO.NET, invoke the `Commit` method on the `Transaction` object:

```
transaction.Commit( );
```

ADO.NET is more object-oriented than many database-programming APIs, as can be seen from the encapsulation of a transaction into a unique object type. This has the drawback of making the methods for explicitly ending a transaction more difficult to find than those to begin a transaction, since they are on different object types. Remember that the method to begin a transaction is on the `Connection` object, and the methods to commit or roll back a transaction are found on the `Transaction` object.

5.4.2.2 Committing a JDBC transaction

Invoking the `commit` method on a JDBC `Connection` object commits a transaction within JDBC and begins a new transaction:

```
connection.commit( );
```

If the connection is in `AUTO COMMIT` mode, you will then get a Java exception of type `SQLException` because the connection will not have a pending transaction.

5.4.3 Rolling Back a Transaction

Rolling back a transaction is a way to close the transaction explicitly and to discard any database modifications since the transaction was started.

5.4.3.1 Rolling back an ADO.NET transaction

To roll back a transaction using ADO.NET, invoke the `Rollback` method on the `Transaction` object:

```
transaction.Rollback( );
```

After rolling back a `Transaction` object, the `Dispose` method should be invoked to free any resources held by the

transaction:

```
transaction.Dispose( );
```

5.4.3.2 Rolling back a JDBC transaction

Invoking the `rollback` method on a JDBC `Connection` object rolls back a transaction within JDBC and begins a new transaction:

```
connection.rollback( );
```

If the connection is in `AUTO COMMIT` mode, then a Java exception will be thrown of `SQLException` type.



< Day Day Up >



5.5 Executing Statements

The primary purpose of programming to a specific database API is to execute SQL statements. The next section covers the steps required to successfully execute a simple SQL statement, such as *INSERT* or *UPDATE*, that does not return results. The section following focuses on the more complicated task of executing statements that do return results.

5.5.1 Executing an ADO.NET Statement

The following code fragment provides the syntax for executing a SQL *INSERT* statement using ADO.NET:

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
statement.CommandText = "INSERT INTO authors(au_id, au_lname, " +  
    "          au_fname, contract) " +  
    "VALUES ('xyz', 'Brown', 'Emmit', 1)";  
int rowsInserted = statement.ExecuteNonQuery( );  
statement.Close( );
```

To execute a statement using ADO.NET, take the following steps:

1. Create an ADO.NET **Command** object. **SqlCommand** is used for access to Microsoft SQL Server, **OdbcCommand** is used for ODBC data sources, and **OleDbCommand** for OLE-DB data sources.

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );
```

2. After creating the appropriate **Command** object, a SQL statement needs to be associated with the **Command** before it can be executed.

```
statement.CommandText = "INSERT INTO authors(au_id, au_lname, " +  
    "          au_fname, contract) " +  
    "VALUES ('xyz', 'Brown', 'Emmit', 1)";
```

3. After assignment of the SQL statement, the **Command** object is ready for execution.

```
int rowsInserted = statement.ExecuteNonQuery( );
```

There are three methods that can be used to execute ADO.NET **Command** objects:

- a. **ExecuteReader**: Executes SQL statements that return rows, such as *SELECT* statements. The return value is an *ADO.NET DataReader* object.
- b. **ExecuteNonQuery**: Executes SQL statements that do not typically return result sets, such as *INSERT*, *DELETE*, or *UPDATE* statements. The return value is an integer value equal to the number of rows affected by the statement execution.
- c. **ExecuteScalar**: Executes SQL statements that return a single value, such as statements containing a single aggregate function.
- d. After the **Command** object is executed, it should be explicitly closed to free up resources that have been allocated to execute statements and to communicate with the database server. To free up these resources, invoke the **Command** object's **Dispose** method:

```
statement.Dispose( );
```



The resources held by ADO.NET **Statement** objects are freed using the **Dispose** method, which differs from the **Close** method used by all other ADO.NET types.



5.5.2 Executing a JDBC Statement

The following code fragment provides the syntax for executing a SQL *INSERT* statement using JDBC:

```
java.sql.Statement statement = connection.createStatement( );  
  
int rowsInserted = statement.executeUpdate(  
    "INSERT INTO authors(au_id, au_lname, au_fname, contract) " +  
    " VALUES ('xyz', 'Brown', 'Emmit', 1)");  
  
statement.close( );
```

1. Executing a SQL statement first requires the creation of a JDBC **Statement** object. A **Statement** object is produced by invoking the `createStatement` method on a valid JDBC **Connection** object:

```
java.sql.Statement statement = connection.createStatement( );
```

2. After a **Statement** object has been created, a SQL statement can be executed by invoking one of the execute methods found on the **Statement** object. For non-query statements, the `executeUpdate` method is the best method to use:

```
statement.executeUpdate("INSERT INTO authors(au_id, au_lname, " +  
    "au_fname, contract) VALUES " +  
    "('xyz', 'Brown', 'Emmit', 1)");
```

The `executeUpdate` method returns an `int` value indicating the number of rows inserted, updated, or deleted during the execution of the SQL statement. If the execution fails, you will get a **SQLException**.

3. After the **Statement** object has been executed, it may be executed again, or freed by invoking the `close` method.

```
statement.close( );
```



5.6 Retrieving Data

A typical database program retrieves data from a database server and processes it. These programs will execute *SELECT* statements following a process similar to that described in the previous section. The difference between executing a *SELECT* statement as opposed to a statement that does not return results is that with *SELECT* you must execute additional code to process the results returned by *SELECT*.

5.6.1 Retrieving Data Using ADO.NET

The following C# code fragment executes a SQL *SELECT* statement that returns author names from the **authors** table and then iterates through the results one row at a time, printing out each author's name. The code is explained in the detailed steps that follow:

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
  
statement.CommandText = "SELECT au_fname, au_lname FROM authors";  
  
{Odbc|OleDb|Sql}DataReader resultSet = statement.ExecuteReader( );  
  
while( resultSet.Read( ) )  
{  
    String fname = "NULL";  
    String lname = "NULL";  
  
    if( !resultSet.IsDBNull( 0 ) ) fname = resultSet.GetString( 0 );  
    if( !resultSet.IsDBNull( 1 ) ) lname = resultSet.GetString( 1 );  
  
    System.Console.WriteLine( lname + ", " + fname );  
}  
  
resultSet.Close( );  
statement.Close( );
```



Column ordinals are all zero-based (the first column is 0, the second is 1, etc.) in ADO.NET, which is different from the one-based ordinals used by JDBC.

To execute a SQL query and process the results using ADO.NET, take the following steps:

1. Create the **Command** object that will be used to execute the *SELECT* statement and attach a *SELECT* statement to it:

```
{Odbc|OleDb|Sql}Command statement = connection.CreateCommand( );  
  
statement.CommandText = "SELECT au_fname, au_lname FROM AUTHORS";
```

2. Invoke the **ExecuteReader** method on the **Command** object, creating a new **DataReader** object.

```
{Odbc|OleDb|Sql}DataReader resultSet = statement.ExecuteReader( );
```

3. Iterate through each row in the result set. After execution of the *SELECT* statement, retrieve rows using the **DataReader's Read** method. If you expect a multiple row result, you should invoke the **Read** method from within a while loop:

```
while( resultSet.Read( ) )
{
```

4. Having fetched a row of data into the **DataReader** object, the column data can be extracted using the **DataReader's Get** methods. Before calling the **Get** methods, we check to see if the data is NULL using the **DataReader's IsDBNull** method. If the value returned by the **IsDBNull** method is true, then the value of the string will be NULL. There are many **Get** methods for each type of column. Using the appropriate datatype for the column is important, because a conversion is not always possible. For a list of available **Get** methods and the ANSI SQL types they should be used with, please see [Table 5-4](#).

```
String fname = "NULL";

String lname = "NULL";

if( !resultSet.IsDBNull( 0 ) ) fname = resultSet.GetString( 0 );

if( !resultSet.IsDBNull( 1 ) ) lname = resultSet.GetString( 1 );
```

When creating an ADO.NET program that fetches data that can contain NULL values, it is always safest to check the return value of the **DataReader** object's **IsDBNull** method prior to extracting a value.

5. The **Read** method of **DataReader** returns false when all the data has been read from the database server. At that point, the **Close** methods on the **Command** object and **DataReader** should be invoked to free up the resources used internally to process the statement.

```
resultSet.Close( );
```

```
statement.Close( );
```

Table 5-4. ADO.NET DataReader Get methods

Method name	Description
GetBoolean(int i)	Returns the value from the <i>i</i> th column as a Boolean value, where <i>i</i> is the zero-based column number.
GetByte(int i)	Returns the value from the <i>i</i> th column as a single-byte value, where <i>i</i> is the zero-based column number. No conversion will be done if the data in the column exceeds one byte and an InvalidCastException object will be thrown.
GetBytes(int i, long dataIndex, byte[] buffer, int bufferIndex, int length)	Returns the value from the <i>i</i> th column as a binary value, where <i>i</i> is the zero-based column number, <i>dataIndex</i> is the offset in the column value to start reading, <i>buffer</i> is the byte array to copy the data into, <i>bufferIndex</i> is the offset into <i>buffer</i> at which to start copying data, and <i>length</i> is the maximum length to copy into <i>buffer</i> .
GetChar(int i)	Returns a single char value from the <i>i</i> th column. For use with character type columns, where <i>i</i> is the zero-based column number.
GetChars(int i, long dataIndex, char[] buffer, int bufferIndex, int length)	Returns a string of characters from the <i>i</i> th column. For use with character type columns, where <i>i</i> is the zero-based column number, <i>dataIndex</i> is the offset in the column value to start reading, <i>buffer</i> is the char array to copy the data into, <i>bufferIndex</i> is the offset into <i>buffer</i> at which to start copying data, and <i>length</i> is the maximum length to copy into <i>buffer</i> .
GetDataTypeName(int i)	Gets a String value that contains the name of the column's datatype, where <i>i</i> is the zero-based column number.
GetDateTime(int i)	Returns a DateTime value from the <i>i</i> th column. For use with temporal type columns, where <i>i</i> is the zero-based column number.
GetDecimal(int i)	Returns a single Decimal value from the <i>i</i> th column. For use with numeric type columns, where <i>i</i> is the zero-based column number.
GetDouble(int i)	Returns a single double value from the <i>i</i> th column. For use with double precision type columns, where <i>i</i> is the zero-based column number.
GetFloat(int i)	Returns a single float value from the <i>i</i> th column. For use with floating-point columns, where <i>i</i> is the zero-based column number.
GetInt{16,32,64}(int i)	Retrieves data from integer columns. The precision of the return value is encoded within the function names. Use GetInt16 for a 16-bit signed short integer, GetInt32 for a 32-bit signed int , and GetInt64 for a 64-bit signed long .

<code>GetName(int i)</code>	Returns a String value containing the name of the <i>i</i> th column, where <i>i</i> is the zero-based column number.
<code>GetOrdinal(string name)</code>	Returns an int value containing the ordinal value of the column with name matching the <i>name</i> argument.
<code>GetString(int i)</code>	Returns a single String value from the <i>i</i> th column. For use with character type columns, where <i>i</i> is the zero-based column number.

Beyond the **Get** methods, there are three other methods on the **DataReader** type that are frequently used when processing data from a query. Those three methods are listed in [Table 5-5](#).

Table 5-5. Frequently used ADO.NET DataReader methods

Method name	Description
<code>Close()</code>	Closes the DataReader object, freeing up resources held by the instance.
<code>IsDBNull(int i)</code>	Returns true if the specified column is NULL, otherwise it returns false, where <i>i</i> is the zero-based column number.
<code>Read()</code>	Fetch the next row if one is available, and return true, otherwise return false.

5.6.2 Retrieving Data Using JDBC

The following Java code fragment executes a SQL *SELECT* statement that returns author names from the **authors** table and then iterates through the results one row at a time, printing out each author's name:

```
java.sql.Statement statement = connection.createStatement();

java.sql.ResultSet result =

    statement.executeQuery("SELECT au_fname, au_lname FROM authors");

while( result.next() ) {

    String fname = result.getString( 1 );

    if( result.wasNull() ) fname = "NULL";

    String lname = result.getString( 2 );

    if( result.wasNull() ) lname = "NULL";

    System.out.println( lname + ", " + fname );

}

result.close();

statement.close();
```



Column ordinals are all one-based (first column is 1, second is 2, etc.) in JDBC, which is different from the zero-based ordinals used by ADO.NET.

5.6.2.1 Use the following steps to execute query statements in JDBC:

1. Create a JDBC **Statement** object by invoking the `createStatement` method on a valid **Connection** object:

```
java.sql.Statement statement = connection.createStatement();
```


- The query is executed by invoking one of the `execute` methods on the `Statement` object. Result sets from query statements are processed by JDBC `ResultSet` objects, which are returned from the `executeQuery` method of JDBC `Statement` objects.

```
java.sql.ResultSet result =  
  
    statement.executeQuery("SELECT au_fname, au_lname FROM authors" );
```

- After creating a `ResultSet` object, you can iterate through one row at a time by invoking the `next` method. The `next` method returns a Boolean value, true for each row in a result set and false after all rows have been iterated through. It is common to invoke the `next` method within a `while` loop to process the rows one-by-one. The `ResultSet` does not begin on the first row of the result, so you must invoke the `next` method to advance to the first row before calling any of the `get` methods.

```
while( result.next( ) ) {
```

- To retrieve column data from the rows within a result set, invoke the appropriate `get` method on the `ResultSet` object. For a list of the most common `get` methods, check [Table 5-6](#). Note that the column data is checked for a NULL value after the `get` method has been invoked, since the nullness of a value can't be determined from the value returned by the `get` methods.

```
String fname = result.getString("au_fname");  
  
if( result.isNull( ) ) fname = "NULL";  
  
String lname = result.getString("au_lname");  
  
if( result.isNull( ) ) lname = "NULL";
```

When creating a JDBC program that fetches data from nullable columns, it is always safest to check the value returned from a `ResultSet` object for a NULL value using the `isNull` method.



The JDBC `get` methods that return object types will return Java NULL values when returning database NULL values. However, this does not apply to non-object types such as `getInt()`, which returns zero in the case of a database NULL. For this reason, this chapter uses the verbose `isNull()` method to test the value for a NULL.

- Free the resources held by the result set and statement objects. When finished with the `ResultSet` and `Statement` objects, invoke their `close` methods so that database resources can be freed.

```
result.close( );  
  
statement.close( );
```

Table 5-6. JDBC ResultSet get methods

Method name	Description
<code>getBlob({int i/String name})</code>	Retrieves a <code>Blob</code> value from <code>BLOB</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getBoolean({int i/String name})</code>	Retrieves a <code>boolean</code> value from <code>BOOLEAN</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getByte({int i/String name})</code>	Retrieves a <code>byte</code> value from <code>CHARACTER</code> or <code>BINARY</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getBytes({int i/String name})</code>	Retrieves a <code>byte[]</code> value from <code>BINARY</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getClob({int i/String name})</code>	Retrieves a <code>Clob</code> value from <code>CLOB</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getDate({int i/String name})</code>	Retrieves a <code>Date</code> value from <code>TEMPORAL</code> type columns, where <code>i</code> is the one-based column ordinal and <code>name</code> is the name of the column.
<code>getDouble({int i/String</code>	Retrieves a <code>double</code> value from <code>DOUBLE PRECISION</code> type columns, where <code>i</code> is the one-based

<code>name}</code>)	column ordinal and <i>name</i> is the name of the column.
<code>getFloat({int i String name})</code>	Retrieves a float value from <i>REAL</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
<code>getInt({int i String name})</code>	Retrieves an int value from <i>INTEGER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
<code>getLong({int i String name})</code>	Retrieves a long value from <i>INTEGER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
<code>getRow()</code>	Returns the current row number.
<code>getShort({int i String name})</code>	Retrieves a short value from <i>INTEGER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
<code>getString({int i String name})</code>	Retrieves a String value from <i>CHARACTER</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
<code>getTime({int i String name})</code>	Retrieves a Time value from <i>TEMPORAL</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.
<code>getTimestamp({int i String name})</code>	Retrieves a Timestamp value from <i>TEMPORAL</i> type columns, where <i>i</i> is the one-based column ordinal and <i>name</i> is the name of the column.

Beyond the **get** methods, there are three other methods on the **ResultSet** type that are frequently used when processing data from a query. Those three methods are listed in [Table 5-7](#).

Table 5-7. Frequently used JDBC ResultSet methods

Method name	Description
<code>close()</code>	Closes the ResultSet object, freeing up resources held by the instance.
<code>next()</code>	Advances the ResultSet object to the next available row and returns true. If no rows remain false will be returned.
<code>wasNull()</code>	Returns true if the last column returned with a get method contained a database NULL value, returns false otherwise.

5.7 Bound Parameters

One of the easiest ways to optimize a slow database application that executes many similar SQL statements is to parameterize the most commonly used SQL statements. Parameterization is a way of reusing a SQL statement by writing it with placeholders for frequently changing values. This provides two major benefits: the potential for reduced server roundtrips and better processing efficiency on the database server, since the server doesn't need to parse, plan, and optimize the execution of frequently used SQL statements on every execution.

Parametrized statements are useful in using string or binary values within SQL statements that contain unfriendly characters, such as quote marks or terminal NULL characters that indicate the end of a string. Usage in this way provides security benefits to a database application where user input is used within statements. Not binding a parameter coming from an end user could allow clever modification of the statement in a way that may divulge secret information from the database.



Since there is a slight cost involved in setting up a parameterized statement, it's good practice to only parameterize statements that will be executed at least three times prior to the statement closing. If a statement is only executed once prior to freeing the statement, the costs of preparing the statement and binding the parameters could result in a performance net loss to the application.

5.7.1 ADO.NET Bound Parameters

The following C# code fragment executes a SQL *INSERT* statement that adds new sales to the **sales** table in the **pubs** database. The *INSERT* statement is parameterized to provide better performance, since the statement object needs to be parsed only once on the server.

```
// Create a Command object for the SQL statement
Statement statement = connection.CreateCommand( );

statement.CommandText =
    "INSERT INTO SALES(stor_id,
        ord_num,
        ord_date,
        qty,
        payterms,
        title_id) " +
    "VALUES(@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id)";

//Prepare the statement on the server
statement.Prepare( );

// Declare parameters that will be bound
{Odbc|OleDb|Sql}Parameter stor_id, ord_num, ord_date,
    qty, payterms, title_id;

stor_id = statement.Parameters.Add( "@stor_id", DbType.String );
ord_num = statement.Parameters.Add( "@ord_num", DbType.String );
ord_date = statement.Parameters.Add( "@ord_date", DbType.DateTime);
```

```
qty = statement.Parameters.Add( "@qty", DbType.Int16 );
payterms = statement.Parameters.Add( "@payterms", DbType.String );
title_id = statement.Parameters.Add( "@title_id", DbType.String );

while( GetNextSale(stor_id, ord_num, ord_date, qty, payterms, title_id )
{
    // Execute the statement
    int result = statement.ExecuteNonQuery( );
    if( result != 1 )
    {
        // If result isn't 1, then the insert failed.
        System.Console.WriteLine( "The INSERT failed." );
        break;
    }
}
```

5.7.1.1 Use the following steps to execute statements with bound parameters in ADONET:

1. As done in previous sections, we create an ADO.NET **Command** object and assign a SQL statement to it. The difference for using bound parameters within the statement is in the *VALUES* clause of the *INSERT* statement. Contained within the *VALUES* clause are six named placeholders for the parameters that will be later bound to the **Command** object. In ADO.NET the placeholders begin with the @ symbol and are followed by an identifier that is unique amongst all placeholders in the statement.

```
Statement statement = connection.CreateCommand( );
statement.CommandText =
    "INSERT INTO SALES(stor_id,
        ord_num,
        ord_date,
        qty,
        payterms,
        title_id) " +
    "VALUES(@stor_id, @ord_num, @ord_date, @qty, @payterms, @title_id)";
```

2. Invoke the **Prepare** method on the **Command** object to prepare the SQL statement for execution. This notifies the database layer that the **Command** object will be executed with bound parameters.

```
statement.Prepare( );
```

3. Declare the parameter objects using the parameter type that matches the **Command** object: **OdbcParameter**, **OleDbParameter**, or **SqlParameter**. After declaring the parameters, create parameter objects by invoking the **Add** method of the **Command** object's **Parameter** collection and assign the return value to the **Parameter** object. The first argument to the **Add** method is the name of the placeholder that the parameter will map to during execution. In the example below, the **stor_id Parameter** object will map to the first item in the *VALUES* clause, which has the placeholder named **@stor_id**. The second argument in the **Add** method is the column type that the placeholder maps to on the server. Look to [Table 5-8](#) for a list of frequently used types.

```
{Odbc|OleDb|Sql}Parameter stor_id, ord_num, ord_date,  
    qty, payterms, title_id;  
  
stor_id = statement.Parameters.Add( "@stor_id", DbType.String );  
ord_num = statement.Parameters.Add( "@ord_num", DbType.String );  
ord_date = statement.Parameters.Add( "@ord_date", DbType.DateTime);  
qty = statement.Parameters.Add( "@qty", DbType.Int16 );  
payterms = statement.Parameters.Add( "@payterms", DbType.String );  
title_id = statement.Parameters.Add( "@title_id", DbType.String );
```

4. In this example, the `Parameter` objects are assigned a value by the user-defined `GetNextSale` function call, which could be implemented like this:

```
static bool GetNextSale(SqlParameter stor_id,  
    SqlParameter ord_num,  
    SqlParameter ord_date,  
    SqlParameter qty,  
    SqlParameter payterms,  
    SqlParameter title_id)  
{  
    // Omitted is the code that would  
    // Read a sale record from a file, or user input, etc.  
  
    // If there are no more sale records, return false.  
    if( !more_records ) return false;  
  
    // Assign values to the parameter objects  
    stor_id.Value = 1234;  
    ord_num.Value = "ABCD.123";  
    ord_date.Value = new DateTime(2003,2,24);  
    qty.Value = 50;  
    payterms.Value = "Net 60";  
    title_id.Value = "SD2043";  
    return true;  
}
```

Notice that the parameters are assigned a value by assigning directly to the `Value` property on the `Parameter` object. The `stor_id` parameter object is assigned a value of 1234, whereas `ord_date` is assigned a C# ADO.NET `DateTime` object. The function returns false if there are no more sales to insert into the table; otherwise the function returns true.

Combined with a `while` loop, the program will continue to insert new sales into the database until the `GetNextSale` function runs out of new records to process.

```
while(GetNextSale(stor_id, ord_num, ord_date, qty, payterms, title_id) )  
{
```

5. Invoking the `ExecuteNonQuery` method on the `Command` object executes the `INSERT` statement, with the bound parameter values replaced for their corresponding placeholders. The `ExecuteNonQuery` method returns the number of rows affected, which will be 1 in this case on a successful single-row insert. This return value is used in error handling and the application will exit if the statement should ever fail to execute.

```
// Execute the statement  
  
int result = statement.ExecuteNonQuery( );  
  
if( result != 1 )  
{  
    // If result isn't 1, then the insert failed.  
    System.Console.WriteLine( "The INSERT failed." );  
    break;  
}
```

Table 5-8. Frequently used parameter object types

DbType object type	Description
<code>AnsiString</code>	Variable-length, non-Unicode character string between 1 and 8,000 characters.
<code>AnsiStringFixedLength</code>	Fixed-length, non-Unicode character string.
<code>Binary</code>	Variable-length binary string between 1 and 8,000 bytes.
<code>Boolean</code>	Represents a true or false value.
<code>Byte</code>	Unsigned integer value ranging 0 to 255.
<code>Currency</code>	Currency value ranging from -2^{63} to $2^{63}-1$.
<code>DateTime</code>	Date and time value.
<code>Decimal</code>	Numeric value ranging from 10^{-28} to $7.9 \cdot 10^{28}$ with about 28 significant digits.
<code>Double</code>	Floating point type ranging from $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$ with about 15 digits.
<code>Int{16,32,64}</code>	Signed integer; the number suffix indicates the number of bits of precision. For example, <code>Int32</code> is a 32-bit integer.
<code>Single</code>	Floating point type ranging from $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$, with about 15 digits.
<code>String</code>	Variable-length, Unicode character string.
<code>StringFixedLength</code>	Fixed-length, Unicode character string.
<code>UInt{16,32,64}</code>	Unsigned integer; the number suffix indicates the number of bits of precision. For example, <code>UInt32</code> is a 32-bit unsigned integer.

5.7.2 Binding Parameters with JDBC

The following Java code fragment executes a SQL `INSERT` statement that adds new sales to the `sales` table in the `pubs` database. The `INSERT` statement is parameterized to provide better performance.

```
// Create a Command object for the SQL statement  
  
PreparedStatement statement = connection.prepareStatement(
```

```
"INSERT INTO SALES(stor_id,
                    ord_num,
                    ord_date,
                    qty,
                    payterms,
                    title_id) " +
"VALUES(?, ?, ?, ?, ?, ?)";

while( getNextSale(statement) )
{
    // Execute the statement
    int result = statement.executeUpdate( );
    if( result != 1 )
    {
        // If result isn't 1, then the insert failed.
        System.out.println( "The INSERT failed." );
        break;
    }
}
```

5.7.2.1 Use the following steps to execute statements with bound parameters in JDBC:

1. Create a JDBC `PreparedStatement` object and pass the parameterized SQL statement into its constructor. The difference for using bound parameters within the statement is in the `VALUES` clause of the `INSERT` statement. Contained within the `VALUES` clause are six placeholders (the question marks) for the parameters that will later be bound to the `PreparedStatement` object.

```
PreparedStatement statement = connection.prepareStatement(
    "INSERT INTO SALES(stor_id,
                      ord_num,
                      ord_date,
                      qty,
                      payterms,
                      title_id) " +
    "VALUES(?, ?, ?, ?, ?, ?)");
```

1. In this example, the parameters are assigned a value by the user-defined `getNextSale` function call, which could be implemented like this:

```
static boolean getNextSale( PreparedStatement statement )
```

```
throws SQLException
{
    // Omitted is the code that would
    // Read a sale record from a file, or user input, etc.

    // If there are no more sale records, return false.
    if( !more_records ) return false;

    statement.setString(1, "1234");
    statement.setString(2, "ABCD.123");
    statement.setDate(3, new java.sql.Date(2003, 2, 24));
    statement.setInt(4,50);
    statement.setString(5, "Net 60");
    statement.setString(6, "SD2043");

    return true;
}
```

1. Each binding position is referenced by its ordinal position in the SQL statement, with the first position starting at 1. The values are assigned to placeholders using the `set` methods found on the `PreparedStatement` object. [Table 5-9](#) contains a list of frequently used `set` methods.
2. The function returns false if there are no more sales to insert into the table; otherwise the function returns true.
3. Combined with a `while` loop, the program will continue to insert new sales into the database until the `getNextSale` function runs out of new records to process.

```
while( getNextSale(statement) )
{
```

1. Invoking the `executeUpdate` method on the `PreparedStatement` object executes the `INSERT` statement, with the bound parameter values replaced for their corresponding placeholders. The `executeUpdate` method returns the number of rows affected, which will be 1 in this case on a successful single-row insert. This return value is used in error handling and the application will exit if the statement should ever fail to execute.

```
    // Execute the statement

    int result = statement.executeUpdate( );

    if( result != 1 )
    {
        // If result isn't 1, then the insert failed.

        System.out.println( "The INSERT failed." );

        break;
    }
}
```

Table 5-9. Frequently used PreparedStatement set methods

Method name	Description
-------------	-------------

<code>setBlob(int i, Blob value)</code>	Sets the placeholder at position <i>i</i> to the Blob contained in <i>value</i> .
<code>setBoolean(int i, boolean value)</code>	Sets the placeholder at position <i>i</i> to the boolean contained in <i>value</i> .
<code>setByte(int i, byte value)</code>	Sets the placeholder at position <i>i</i> to the byte contained in <i>value</i> .
<code>setClob(int i, Clob value)</code>	Sets the placeholder at position <i>i</i> to the Clob contained in <i>value</i> .
<code>setDate(int i, Date value[, Calendar cal])</code>	Sets the placeholder at position <i>i</i> to the Date contained in <i>value</i> . If <i>cal</i> is provided, it'll be used to interpret the Date value .
<code>setDouble(int i, double value)</code>	Sets the placeholder at position <i>i</i> to the double contained in <i>value</i> .
<code>setFloat(int i, float value)</code>	Sets the placeholder at position <i>i</i> to the float contained in <i>value</i> .
<code>setInt(int i, int value)</code>	Sets the placeholder at position <i>i</i> to the int contained in <i>value</i> .
<code>setLong(int i, long value)</code>	Sets the placeholder at position <i>i</i> to the long contained in <i>value</i> .
<code>setNull(int i, int v)</code>	Sets the placeholder at position <i>i</i> to a NULL value when <i>v</i> is true.
<code>setString(int i, String value)</code>	Sets the placeholder at position <i>i</i> to the String contained in <i>value</i> .
<code>setTimestamp(int i, Timestamp value[,Calendar cal])</code>	Sets the placeholder at position <i>i</i> to the Timestamp contained in <i>value</i> . If <i>cal</i> is provided, it'll be used to interpret the Timestamp value .

5.8 Error Handling

With everything that can go wrong in a database application, the correct handling of errors is a topic large enough for its own chapter (or book!). We're lucky that the APIs covered in this book, built on the modern Java language and on the .NET framework, contain error handling mechanisms that make recovering from errors easy.

5.8.1 Error Handling in ADO.NET

ADO.NET's error handling model is based on .NET exceptions. Functions or methods that can fail because of an error must be wrapped in a `try/catch/finally` block. The difficult part of handling exceptions is determining what to do when an error occurs and freeing resources for objects that were created. The following code example shows the error handling pattern that can be successfully used with ADO.NET to clean up all resources in the event of an error:

```
{Odbc|OleDb|Sql}Connection connection = null;

{Odbc|OleDb|Sql}Command statement = null;

{Odbc|OleDb|Sql}DataReader resultSet = null;

try {

    // Create and use the Connection, Command, and DataReader objects

    connection = new SqlConnection(connection_string);

    connection.Open( );

    statement = connection.CreateCommand( );

    statement.CommandText = SQL;

    resultSet = statement.ExecuteReader( );

} catch( Exception e ) {

    // Notify user that an error occurred

} finally {

    if( resultSet != null ) resultSet.Close( );

    if( statement != null ) statement.Dispose( );

    if( connection != null ) connection.Close( );

}
```

If an error occurs in the `try` block and an exception (of type `Exception` in this case) is thrown, the exception will be caught and the code in the `catch` block will be executed. The `finally` block, which will be executed regardless of whether an exception is thrown, should be responsible for freeing the resources. Which resources to free is difficult to determine if you don't know the point of failure. For example, if the failure occurs when executing a SQL statement, then the only resources that need to be freed are the `statement` and `connection` objects. In the code fragment above, an exception thrown in the `ExecuteReader` method would prevent the `resultSet` object from being closed, since it would still be set to its initial value of `null`.



It is good programming practice to free resources in the opposite order in which they were allocated.

5.8.2 Error Handling in JDBC

JDBC's error handling model is based on Java exceptions; therefore, the standard **try/catch/finally** error handling paradigm of Java can be applied to JDBC applications. The following Java code fragment is an example of how to use a **try/catch/finally** code block to gracefully handle application exceptions and free allocated resources:

```
Connection connection = null;

Statement statement = null;

ResultSet resultSet = null;

try {

    // Create and use the Connection, Statement, and ResultSet objects

    connection = DriverManager.getConnection( connection_string );

    statement = connection.createStatement( );

    resultSet = statement.executeQuery( SQL );

} catch( Exception e ) {

    // Then, notify the user of an error.

} finally {

    if( resultSet != null )

        try{resultSet.close( );} catch(Exception e) {}

    if( statement != null )

        try{statement.close( );} catch(Exception e) {}

    if( connection != null )

        try{connection.close( );} catch(Exception e) {}

}
```

If an error occurs in the **try** block and an exception (of type **Exception** in this case) is thrown, the exception will be caught and the code in the **catch** block will be executed. The **finally** block, which will be executed regardless of whether an exception is thrown, should be responsible for freeing the resources held by the **connection**, **result set**, and **statement** objects. Note that each object is first tested for a null value before freeing the resources held by the object. This NULL test is to make the code applicable to all error conditions — the code functions if the exception occurs while opening the connection as well as processing the **result set** data. The **close** methods on the JDBC objects can also generate exceptions when freeing resources, so they must be wrapped in their own **try/catch** blocks. This makes coding awkward, but even so, it will be relatively simple if you follow the pattern given here.



It is good programming practice to free resources in the opposite order from what they were allocated.

5.9 Examples

In the previous sections of this chapter, we examined the fundamental components of SQL statement processing found in most database programs, but didn't include any programs in their entirety. In the following section, we combine the necessary steps into a small program that executes a simple *SELECT* statement and prints out the results. The same example is provided for both of the APIs covered in this chapter.

The examples execute a *SELECT* statement and print out the results after establishing connectivity with the database. The *SELECT* statement is as follows:

```
SELECT a.au_lname, a.au_fname, SUM(t.ytd_sales)
FROM authors a, titleauthor, titles t
WHERE titleauthor.au_id = a.au_id and
      titleauthor.title_id = t.title_id
GROUP BY a.au_lname, a.au_fname
ORDER BY 3 DESC
```

The examples execute the statement and print out the three-column result set returned by the database server.

5.9.1 ADO.NET Example

The following C# ADO.NET database program connects to a database and prints a list of authors in the **pubs** database that includes their year-to-date sales. This program can be easily adapted to meet other database processing needs by following the practices outlined in the earlier sections of this chapter.

```
using System;
using System.Data.SqlClient;

class ExampleApplication
{
    static void Main(string[] args)
    {
        String connection_string =
            "Server=(local);Trusted_Connection=true;DATABASE=pubs;";
        String SQL =
            "SELECT a.au_lname, a.au_fname, SUM(t.ytd_sales) " +
            "FROM authors a, titleauthor, titles t " +
            "WHERE titleauthor.au_id = a.au_id and " +
            "      titleauthor.title_id = t.title_id " +
            "GROUP BY a.au_lname, a.au_fname " +
            "ORDER BY 3 DESC";

        SqlConnection connection = null;
        SqlCommand statement = null;
        SqlDataReader resultSet = null;
```

```
try
{
    // Create Connection and Connect to the Server
    connection = new SqlConnection(connection_string);
    connection.Open( );

    // Create a Command object for the SQL statement
    statement = connection.CreateCommand( );
    statement.CommandText = SQL;

    // Create a Reader for reading the result set
    resultSet = statement.ExecuteReader( );
    while( resultSet.Read( ) )
    {
        // Extract the data from the server and display it
        String fname = "NULL";
        String lname = "NULL";
        String sales = "ZERO";
        if( !resultSet.IsDBNull( 0 ) )
            fname = resultSet.GetString( 0 );
        if( !resultSet.IsDBNull( 1 ) )
            lname = resultSet.GetString( 1 );
        if( !resultSet.IsDBNull( 2 ) )
            sales = resultSet.GetInt32( 2 ).ToString( );
        System.Console.WriteLine( lname + ", " +
            fname + " has sales of " +
            sales);
    }
}

catch( SqlException e )
{
    // Print out the error string, if any.
    System.Console.WriteLine("Error:" + e.ToString( ) );
} finally {
    // Free up resources
    if( resultSet != null ) resultSet.Close( );
    if( statement != null ) statement.Dispose( );
}
```

```
        if( connection != null ) connection.Close( );  
    }  
}  
}
```

5.9.2 JDBC Example

The following Java JDBC database program connects to a database and prints out each of the authors in the **pubs** database, as well as their year-to-date sales.

```
import java.sql.*;  
  
public class ExampleApplication  
{  
    public static void main(String[] args)  
    {  
        String connection_string =  
            "jdbc:microsoft:sqlserver://localhost:1433;" +  
            "User=montoyaj;Password=12345;DatabaseName=pubs;";  
  
        String SQL =  
            "SELECT a.au_lname, a.au_fname, SUM(t.ytd_sales) " +  
            "FROM authors a, titleauthor, titles t " +  
            "WHERE titleauthor.au_id = a.au_id and " +  
            "    titleauthor.title_id = t.title_id " +  
            "GROUP BY a.au_lname, a.au_fname " +  
            "ORDER BY 3 DESC";  
  
        Connection connection = null;  
        Statement statement = null;  
        ResultSet resultSet = null;  
  
        try  
        {  
            Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");  
  
            // Create Connection and Connect to the Server  
            connection = DriverManager.getConnection( connection_string );  
  
            // Create a Command object for the SQL statement
```

```
statement = connection.createStatement( );

// Create a Reader for reading the result set
resultSet = statement.executeQuery( SQL );
while( resultSet.next( ) )
{
    // Extract the data from the server and display it
    String lname = resultSet.getString( 1 );
    if( resultSet.isNull( ) ) lname = "NULL";
    String fname = resultSet.getString( 2 );
    if( resultSet.isNull( ) ) fname = "NULL";
    String sales = resultSet.getString( 3 );
    if( resultSet.isNull( ) ) sales = "ZERO";
    System.out.println( lname + ", " +
        fname + " has sales of " + sales);
}

}

catch( Exception e )
{
    // Print out the error string, if any.
    System.out.println("Error:" + e.toString( ) );
} finally {
    // Free up resources
    if( resultSet != null )
        try {resultSet.close( );} catch( Exception e ) {}
    if( statement != null )
        try {statement.close( );} catch( Exception e ) {}
    if( connection != null )
        try {connection.close( );} catch( Exception e ) {}
}

}

}
```

Chapter 5. Database Programming

While SQL plays an important role in standardizing the communication with different RDBMSs, a missing piece still remains for those who want to write database software applications. The missing piece is the database programming Application Programming Interface (API) used to transport SQL statements to and process their results from an RDBMS. While all the database platforms discussed in this book provide their own proprietary interfaces for database application developers, this chapter focuses on two commonly used APIs that provide a consistent interface across database platforms. Specifically, this chapter introduces you to:

ADO.NET

ADO.NET is Microsoft's high-level database programming API on the .NET platform. The ADO.NET API is a collection of .NET interfaces that are accessible from any of the .NET languages. The primary benefits of ADO.NET are ease of use, portability within the .NET platform, XML integration, and access to data sources other than relational databases. The ADO.NET examples covered in this book are written in C#; however, ADO.NET is also available from Visual Basic and the other .NET languages.

JDBC

JDBC, or *Java Database Connectivity*, was developed primarily by Sun Microsystems to be the primary database programming API for the Java language. JDBC is the most popular database programming API for the Java language and offers operating-system portability, reasonable performance for most applications, and a well-documented interface. In addition, drivers for most database systems are typically free. This chapter covers JDBC Version 3.0. For additional information, please browse <http://java.sun.com/jdbc>.

As a quick desk reference, this chapter won't provide all the information needed to develop a large enterprise database application. However, we give you enough to get started by covering components that are common to all database applications, both large and small.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *SQL in a Nutshell* is a chameleon. There are approximately 85 species of chameleons existing in the world today. They are mostly indigenous to Africa, although there are a few species found in Asia and in Europe. Most are tree dwellers. The chameleon is relatively small; the average adult size is between 6 and 12 inches. It lives mostly on insects and uses its long tongue to capture its prey. Indeed, the tongue is a critical tool. It can stretch up to 1.5 times the lizard's body length, and there is an adhesive pad on the end on which the insects are trapped. There are several other characteristics common to all species of chameleons. For example, its eyes are large and protruding, and the lizard can see 360 degrees without moving its head or body. Its toes are on either side of its feet, usually three on one side and two on the other. This is ideal for moving quickly and efficiently through tree branches.

Chameleons are best known for their ability to change their appearance to adapt to their physical environment. Actually, several types of reptiles can change their skin color, but the chameleon is far and away the most accomplished. This skill, which is moderated by the nervous system, obviously is invaluable for hunting prey and avoiding predators, and also helps to stabilize body temperature. The extent of this camouflage capability is related to the gender, age, and species of the lizard.

Mary Brady was the production editor for *SQL in a Nutshell*, and Jane Ellin was the copyeditor. Ann Schirmer was the proofreader. Sarah Sherman and Colleen Gorman provided quality control. Julie Hawks wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Julie Hawks to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Mary Brady.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.



Copyright © 2004 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

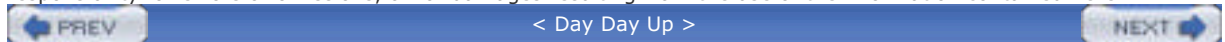
Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The In a Nutshell series designations, *SQL in a Nutshell*, the image of a chameleon, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



Why This Book?

The primary source of information for relational databases is the documentation and help files provided by the vendors themselves. While each vendor's documentation is an indispensable resource that most database programmers and database administrators turn to first, it has a number of limitations:

- It covers only a single, specific vendor's product. There is no coverage for translation, migration, or integration issues.
- It describes programming methods, usually in a multitude of small, disconnected documents or help files.
- It covers individual commands, often in confusing detail, obscuring the simple and direct uses of commands that are used every day by programmers and administrators.

In other words, the documentation included with the vendor database is an exhaustive explanation of every aspect of their platform. After all, help texts are naturally geared toward delivering the main facts about the product. They'll tell you a command's specific syntax (and all its obscure variants) and, in general terms, how to implement it. However, if you move between RDMS's and you need to be productive very quickly, you will rarely use those obscure command variations and instead utilize the common capabilities in real-life situations.

This book begins where the vendor documentation ends by distilling the experiences of professional database administrators and developers who have used these SQL variants day in and day out to support complex enterprise applications. You'll get the benefit of their experience in a compact and easily usable format. Whether SQL is new to you or you have been using SQL since its earliest days, there are always new tips and techniques to learn. And, when moving between different implementations, it's always important to find out about the implementations that can bite you if you're not careful and informed.

Acknowledgments

We'd like to take a moment to thank a few special individuals at O'Reilly Media. First, we owe a huge debt of gratitude to Jonathan Gennick, the editor of the second edition. Jonathan kept us on track and on task, even when the cares of this world threatened to derail the project. Jonathan's attention to detail and exceptional management skills, in addition to his talents as an editor, are the reason this book is here today. Thank you! And of course, thanks to Tim O'Reilly for having a direct hand in the birth of this book.

We also owe a debt to our fine technical reviewers. To Peter Gulutzan (SQL Standard), Thomas Lockhart (PostgreSQL), Alan Beaulieu (Oracle), Baya Pavliachvili (Microsoft SQL Server), Bobby Fielding (DB2), Doug Doole (DB2), Rick Swagerman (DB2), Josh Stellan (DB2), Brian Lalonde (Database Programming), John Haydu (Oracle, Phani Arega (Oracle), and Paul DuBois (MySQL): we owe you a hearty thanks! Your contributions have greatly improved the accuracy, readability, and value of this book. Without you, our sections on each of the language extensions would have been on shaky ground. In addition, we'd like to tip our hat to Peter Gulutzan and Trudy Pelzer for their book *SQL-99 Complete, Really!*, which assisted our understanding of the ANSI SQL2003 standards.

Brand Hunt's Acknowledgments

To my wife Michelle, without your continued support and love, I wouldn't be a part of this project. I'm appreciative for every moment we've shared and your forgiveness for my keeping you awake at night with the "tappity-tap-taps" emanating from the computer.

To my parents Rex and Jackie, the two biggest influences in everything I've ever done correctly — especially those things that frequently take multiple attempts (like writing!)

A huge thanks to my teammates Kevin, Daniel, and Jonathan for letting me participate in this project and exercising so much patience tutoring a first time O'Reilly author. Your professionalism, work ethic, and ability to make the most tedious tasks fun is so admirable I plan to steal it and adopt it as my own!

Mad props to my friends and colleagues at Rogue Wave Software, ProWorks, NewCode Technology, and Systems Research and Development, for providing the ultimate sand box for refining skills in SQL, databases, business, software development, writing, and friendship: Gus Waters, Greg Koerper, Marc Manley, Wendi Minne, Erin Foley, Elaine Cull, Randall Robinson, Dave Ritter, Edin Zulic, David Noor, Jim Shur, Chris Mosbrucker, Dan Robin, Mike Faux, Jason Prothero, Tim Romanowski, Andy Mosbrucker, Jeff Jonas, Jeff Butcher, Charlie Barbour, Steve Dunham, Brian Macy, and Ze'ev Mehler.

Daniel Kline's Acknowledgments

I'd like to thank my brother, Kevin, for his continued willingness to work with me, to my colleagues at the University of Alaska Anchorage for their suggestions, and to the readers and users of the first edition of *SQL in a Nutshell* for their honest feedback and useful critiques. I'd also like to thank the Japanese translators of the first edition—Ryoko Akaike, O'Reilly Japan editor; Ishii Tatsuo, open source manager of Software Research Associates, Inc. and author of *PostgreSQL Perfect Guide*; and Toru Miyahara, CEO of BigiNet Co., Ltd.—for their careful review and helpful suggestions.

Kevin E. Kline's Acknowledgments

Many people helped deliver the big, thick book you hold in your hands. This note expresses our appreciation to those who helped make this a book reality.

First of all, Dan and Brand get a big slap on the back for working so hard to create the content you are now reading. Brand had to spend a lot of nights working on this both before and after his wedding. Thank you, Michelle, for lending us Brand when you wanted him the most.

Next, Jonathan Gennick, our editor at O'Reilly Media, gets a warm handshake and squeeze on the shoulder. You travailed through many extra months of hard labor bringing this book out of the shadows and into the light. I think there's a special blessing in store for those who've carried themselves with as much patience, professionalism, and good humor as you.

We must also express deep thanks and appreciation to our technical reviewers who caught many errors, omissions, and plain ol' mistakes and diverted us from many embarrassments:

- ANSI SQL — Peter Gulutzan, author of *SQL-99 Complete, Really* and *SQL Performance Tuning*
- DB2 — Doug Doole, Bobby Fielding, and Richard Swagerman of IBM; Josh Steffan of Quest Software

- MySQL — Paul DuBois, author of *MySQL*
- Oracle — Phani Arega of Quest Software and the reviewers at Oracle
- PostgreSQL — Thomas Lockhart, original author of the PostgreSQL open source documentation
- SQL Server — Baya Pavliashvili of Healthstream, Inc.
- Database Programming — Brian LaLonde

To all of my colleagues at Quest Software go a very big thanks for your support and encouragement. Rony Lerner, Deb Jenson, Eyal Aronoff, John Newsom, and Vinny Smith: thank you for investing your trust in me and making these last three years with Quest Software such a blast.

Thank you to all the guys on the SQL Server team for making our products the best on the market: John Theron, Patrick O'Keeffe, John Ortega, Mark Simon, Hasan Fahimi, Lee Grissom, Joe Motley, Adrian Tudor, Israel Kalush, Amit Kubovsky, and Ross Doering. I couldn't ask for a better team or a better set of friends.

Thanks to the SQL Server team at Microsoft for keeping me in the know: Euan Garden, Richard Waymire, Joe Yong, Don Peterson, Mark Sousa, Steven Dybing, Fernando Caro, Tom Rizzo, and Bill Baker. Your product rocks!

Finally, a word for my family. It's hard to work ridiculously long hours for months on end, between the day job and the book project, without making some sacrifices. And yet, somehow, we managed to spend so much good time together. Katie Jo, you've learned how to talk between the time I started and the time I finished. I still get choked up hearing you say "Yi yove you, daddy." Anna Lynn, your super powers rescued me plenty of times when I was down. I want to hear "Supergirl to da rescue!" the next time I need rescuing. Here's a secret: I keep that jar of fresh pistachios on my desk just because they're your favorite. Emily, thank you for letting me paint your nails. Some daughters don't let their dads that close. Dylan, you whipped me at so many games on the PS2 that I don't know why you still want me to play with you. But, one day, the ol' man is gonna take you down! Kelly, you are my tonic, my therapy, and my medicine. Just a touch or a glance or a smile from you and my night turns to day and my despair to hope. Thank you for loving me, supporting me, and seeing me through this project.

Who Should Read This Book?

SQL in a Nutshell, Second Edition, benefits several groups of users. It is useful for programmers who require a concise and handy SQL reference tool. It is useful for developers who need to migrate from one SQL dialect to another. Finally, it is useful for database administrators (DBAs) who need to both execute a myriad of SQL statements to keep their enterprise databases up and running, and create and manage objects such as tables, indexes, and views.

This book is a reference work, not a tutorial. The writing is not expository. For example, we won't explain the concept of an elementary loop. Experienced developers already know such things. You want the meat. So we will explain, for example, the detailed workings of an ANSI standard cursor, how it works on each of the database platforms we cover, the special capabilities of cursors on each database platform, and the various pitfalls of cursors and how to get around them.

While we don't intend for *SQL in a Nutshell, Second Edition*, to be a tutorial on SQL or a handbook for database design, we do provide some brief coverage of those topics, and we hope you find that helpful. [Chapter 1](#) and [Chapter 2](#) provide a concise introduction to SQL, covering the general origins, essential structure, and basic operation of the language. If you're new to SQL, the introductions in [Chapter 1](#) and [Chapter 2](#) will help you to get started.

How This Book Is Organized

SQL in a Nutshell, Second Edition, is divided into five chapters and two appendixes:

[Chapter 1](#), *SQL History and Implementations*

Discusses the Relational Database Model, describes the current and previous SQL standards, and introduces the SQL implementations covered in this book.

[Chapter 2](#), *Foundational Concepts*

Describes the fundamental concepts necessary for understanding relational databases and SQL commands.

[Chapter 3](#), *SQL Statement Command Reference*

Provides an alphabetical command reference to SQL statements. It details the latest ANSI standard, SQL2003, for each command, as well as the implementation of each command by DB2, MySQL, Oracle, PostgreSQL, and SQL Server.

[Chapter 4](#), *SQL Functions*

Provides an alphabetical reference of the SQL2003 functions, describing vendor implementations of all SQL2003 functions. In addition, [Chapter 4](#) includes coverage for all platform-specific functions that are unique to each implementation.

[Chapter 5](#), *Database Programming*

Provides an overview of database programming from a variety of connectivity methods. ADO.NET and JDBC are discussed, among other topics.

[Appendix A](#), *Sybase Adaptive Server*

Sybase's Adaptive Server product and Microsoft SQL Server share a common heritage. As a result, most of the SQL Server commands detailed in [Chapter 3](#), and most of the functions detailed in [Chapter 4](#), will work with Sybase. However, "most" is the operative word, and we provide this appendix to show where Sybase differs from SQL Server.

[Appendix B](#), *Shared and Platform-Specific Keywords*

Provides a table of keywords declared in SQL2003 and by the different database platforms. You can use this table to look for words that you should not use for object or variable names.

Conventions Used in This Book

Constant Width

Used to indicate programming syntax, code fragments, and examples.

Constant Width Italic

Used to indicate variables in code that should be replaced with values.

Constant Width Bold

Used in code sections to highlight portions of the code.

Italic

Used to introduce new terms, for emphasis, to indicate commands or user-specified file and directory names, and to indicate variables within text.

Bold

Used to display the names of database objects, such as tables, columns, and stored procedures.

UPPERCASE ITALIC

Used to indicate SQL keywords when they appear in the text.



This icon indicates a tip, suggestion, or general note.



Indicates a warning or caution.



How to Use This Book

SQL in a Nutshell, Second Edition, is primarily a command reference. As a consequence, you'll probably use it to look up a variety of SQL commands and functions. However, with documentation for the ANSI standard itself, plus five database platforms, each command has the potential to get very large.

In order to reduce the verbiage describing each command, we compare each platform's implementation to the SQL2003 standard. If the platform supports a clause described in the SQL2003 discussion, then we won't repeat that clause again.

Generic and transportable examples are provided within the body of each SQL2003 command description. Since the SQL2003 standard is ahead of most database platforms, examples aren't provided for elements of the SQL2003 commands that are not supported by any platform discussed in this book. In addition, more examples are provided for each database platform that highlight unique extensions and enhancements.

We recognize that our approach may necessitate jumping from a description of a platform's implementation of a command back to the corresponding SQL2003 command description. However, we felt that this was better than packing the book with hundreds of pages of redundant content.



How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). We want to hear from you, especially with information that will make this book better. Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

We have a web site for the book, where we'll list any examples, errata, or plans for future editions. You can access this page at:

<http://www.oreilly.com/catalog/sqlnut2/>

Please help us out by pointing out any typos or syntactical errors that you encounter. (You can imagine how hard it is to proofread a book covering the ANSI standard and five separate products.) You may also ask technical questions or comment on the book by sending an email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly web site:

<http://www.oreilly.com>

Safari Enabled



When you see the Safari® Enabled icon on the back cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information.

Try it for free at <http://safari.oreilly.com>.

Resources

The following web sites provide additional information about the various platforms covered in this book:

DB2

IBM's DB2 database is supported on the Web at <http://www.software.ibm.com/data/db2/>. DB2 has an active and vibrant user community located on the Web at <http://www.idug.org>.

MySQL

The corporate resource for MySQL is <http://www.mysql.com>, and another good site is <http://theoryx5.uwinnipeg.ca/mysql>. A great developer resource with lots of useful tips is Devshed.com. See http://www.devshed.com/Server_Side/MySQL/ for MySQL-specific information.

PostgreSQL

The home for this open source database is located at <http://www.postgresql.org>. With a great deal of useful information available for download, this site also maintains mailing lists for PostgreSQL users. Another PostgreSQL site worth investigating is <http://www.pgsql.com>, which offers support for commercial customers.

Oracle

Oracle's cyberspace home is <http://www.oracle.com>. A great resource for hard-core Oracle users is <http://www.oracle.com/technology>. You can also find all Oracle documentation at <http://tahiti.oracle.com>.

SQL Server

The official Microsoft SQL Server web site is <http://www.microsoft.com/sql/>. Another good resource is found at the home of the Professional Association for SQL Server (PASS) at <http://www.sqlpass.org>.

Changes in the Second Edition

Although the ANSI standards committee released the current version of SQL in 2003, called SQL2003, this book has grown dynamically in a variety of new ways in response to request from our readers. In fact, when you compare the first and second editions of *SQL in a Nutshell*, you'll see that the second edition is, in reality, an entirely new book. Here are more details about changes in the second edition:

New format

We've developed an entirely new format that reduces redundancy and increases coverage for the SQL2003 standard and each vendor's implementation. Rather than use the expository style of the first edition, we've taken a keyword description style that breaks all the keywords and clauses into small, bite-sized entries.

New database platforms

We've added full coverage for IBM's DB2 UDB database platform running on Unix, Linux, and Windows. In addition, we've added an appendix that describes Sybase's implementation of the SQL commands, as they differ from Microsoft SQL Server's implementation.

Database programming

We've added a chapter that covers the basics of how programmers can interface their frontend programs to the backend database platforms covered in this book.

More complete coverage

We've added many more examples and added SQL commands not covered in the first edition. In addition, we've added much more complete coverage for SQL functions — in particular, vendor-specific functions that are not part of the SQL2003 standard.

More examples

You can never have too many examples. We've added examples for the most basic uses of the commands under the SQL2003 headings, with even more examples that highlight the unique and powerful extensions offered by each database platform.

Preface

Since its first incarnation in the 1970s, the Structured Query Language (SQL) has been developed hand in hand with the information boom, and as a result, is the most widely used database manipulation language in business and industry. A number of different software companies and program developers, including those in the open source movement, have concurrently developed their own SQL dialects in response to specific professional needs. All the while, standards bodies have developed a growing list of common features.

SQL in a Nutshell, Second Edition, describes the latest ANSI standard, SQL2003, version of each SQL command, and then documents each platform's implementation of that command. In this book, you will find a concise explanation of the Relational Database Management System (RDBMS) model, a clear-cut explanation of foundational RDBMS concepts, and thorough coverage of SQL syntax and commands.

Most importantly, at least if you're a programmer or developer, is that *SQL in a Nutshell*, Second Edition, provides a concise guide both to the most popular commercial database packages on the market (Microsoft SQL Server, IBM's DB2 Universal Database, Sybase Adaptive Server, and Oracle), and to two of the best-known open source (<http://www.opensource.org>) database products (MySQL and PostgreSQL). *SQL in a Nutshell's* attention to open source SQL platforms is an affirmation of the growing importance of the open source movement within the computing community.

The database platforms covered in this book include:

- IBM DB2 UDB Version 8.0 for Linux, Unix, and Windows
- MySQL Version 4
- Oracle Database 10g
- PostgreSQL Version 7.0
- Microsoft SQL Server 2000
- Sybase Adaptive Server Version 12.5



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)
- [Academic](#)

SQL in a Nutshell, 2nd Edition

By [Kevin E. Kline](#)

Publisher: O'Reilly

Pub Date: September 2004

ISBN: 0-596-00481-8

Pages: 710

[Copyright](#)

[Preface](#)

[Why This Book?](#)

[Who Should Read This Book?](#)

[How This Book Is Organized](#)

[Conventions Used in This Book](#)

[How to Use This Book](#)

[How to Contact Us](#)

[Safari Enabled](#)

[Resources](#)

[Changes in the Second Edition](#)

[Acknowledgments](#)

[Chapter 1. SQL History and Implementations](#)

[Section 1.1. The Relational Model and ANSI SQL](#)

[Section 1.2. History of the SQL Standard](#)

[Section 1.3. SQL Dialects](#)

[Chapter 2. Foundational Concepts](#)

[Section 2.1. Database Platforms Described in This Book](#)

[Section 2.2. Categories of Syntax](#)

[Section 2.3. SQL2003 and Platform-Specific Datatypes](#)

[Section 2.4. Constraints](#)

[Chapter 3. SQL Statement Command Reference](#)

[Section 3.1. How to Use This Chapter](#)

[Section 3.2. SQL Platform Support](#)

[Section 3.3. SQL Command Reference](#)

[Chapter 4. SQL Functions](#)

[Section 4.1. Types of Functions](#)

[Section 4.2. ANSI SQL Aggregate Functions](#)

[Section 4.3. ANSI SQL Window Functions](#)

[Section 4.4. ANSI SQL Scalar Functions](#)

[Section 4.5. Platform-Specific Extensions](#)

[Chapter 5. Database Programming](#)

[Section 5.1. Database Programming Overview](#)

[Section 5.2. Opening a Database Connection](#)

[Section 5.3. Closing a Database Connection](#)

[Section 5.4. Managing Transactions](#)

[Section 5.5. Executing Statements](#)

[Section 5.6. Retrieving Data](#)

[Section 5.7. Bound Parameters](#)

[Section 5.8. Error Handling](#)

[Section 5.9. Examples](#)

[Appendix A. Sybase Adaptive Server](#)

[Section A.1. Sybase Adaptive Server Datatypes](#)

[Section A.2. Sybase Adaptive Server SQL Statements](#)

[Section A.3. Sybase Adaptive Server SQL Functions](#)

[Section A.4. Sybase Adaptive Server Keywords](#)

[Appendix B. Shared and Platform-Specific Keywords](#)

[Colophon](#)

[Index](#)