DATABASES

The earliest use of a written language, with agreed symbols standing for ideas, developed in Sumeria in about 3700 BCE. There is evidence that by 3500 BCE temple clerks had started recording wages, tributes, and stores by making impressions on wet clay tablets using a stylus. Precise record keeping by the state and the trader for purposes such as taxation and trade started as soon as the enabling technology of writing was available. In contrast, the first literature is believed to have been developed in 2300 BCE.

Databases managed on digital computers are the modern technology for precise record keeping. The first commercial computer was installed during the 1950s; the first generalized database system, called Integrated Data Store (IDS), was designed at General Electric in 1961 and was in wide distribution by 1964. Database technology makes it possible to store, search, and update large amounts of data quickly. It also makes it possible for multiple users to manipulate the data concurrently while access is limited to authorized users. Further, databases provide some guarantees that the data will not be corrupted or lost because of factors such as user errors and system crashes.

Database technology plays a critical role in almost all computer applications. It is a key component of the infrastructure for the World Wide Web. Databases are used in application areas such as business, engineering, medicine, law, science, the liberal arts, and education. Database software is an important business area and was estimated between 5 billion and 10 billion dollars in 1997.

This article is divided into five sections. The first section provides an introduction to databases and introduces the basic concepts. The next section, "Data Models," describes the fundamental kinds of database systems. The third section, "Transactions and Concurrency Control," describes how a database system guarantees the safety of data and permits concurrent manipulation by multiple users. The fourth section, "System Architecture and Implementation Techniques," describes how a database system answers questions posed against the data. The last section describes some advanced topics.

Basic Concepts

A database is a collection of related data stored on a computer system and accessed by application programs. As an example, consider a hypothetical mail-order company called MOCK that maintains a database of its customers and the orders placed by them. The database will contain data such as the name, address, and phone number of each customer, the parts ordered by each customer, and the status of each order. The

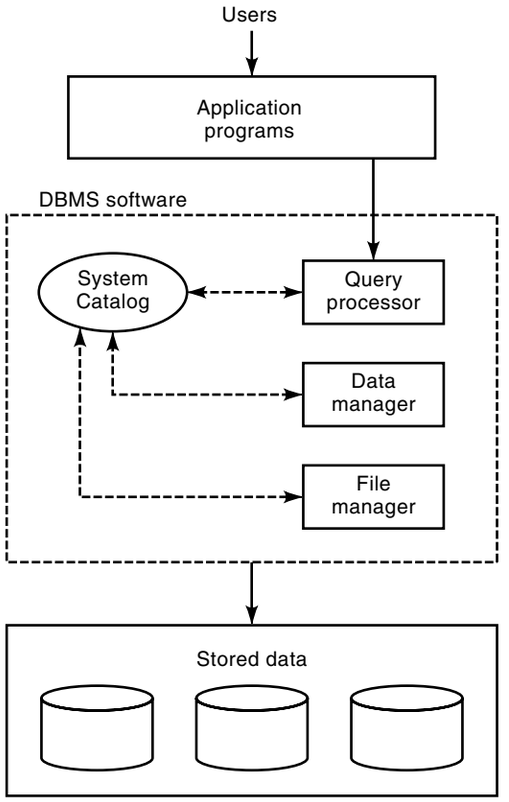


Figure 1. A database system.

data are physically stored on storage devices such as disks and managed by a sofware program called a database management system (DBMS), as shown in Fig. 1. Running a mail-order business requires actions such as adding a new customer, adding a new order for a customer, checking the status of an order, or changing the status of an order when the product is shipped to the customer. Such actions are performed by running application programs that query or modify the database.

When a new customer calls MOCK, the clerk receiving the call will use an application program to store the customer's name, address, and phone number into the database. If the customer orders some merchandise, the clerk will run an order-entry application program. If the customer wants to know the status of an order already placed, the clerk will run an order-status application that finds orders given the name and phone number of a customer. The database is thus used to model and track some aspects of MOCK's business. The DBMS is a general-purpose software that can be used in any application. The specific business needs of MOCK determine the choice of data stored in the database, and some of the actions needed to run the business are encoded as application programs.

Why Use an Electronic Database System?

Use of an electronic database system can lead to dramatic increases in productivity. A single clerk will be able to handle a larger number of customer calls per hour when a database system is used. Once MOCK has more than a few dozen customers, a clerk can run an application program in less time than it takes to find the customer's file in a filing cabinet.

Data in a database system can be easily and correctly shared. Even if clerks are in different cities, it is possible for them to access records at the same time. Even when there is a need for records of the same customer to be accessed simultaneously by two different clerks, a database makes that easily possible and ensures that the data remain consistent.

Data in a database can be easily analyzed to determine how well the business is running. A database system offers a data manipulation language (DML) that permits the data to be updated and questions to be posed against the data. For instance, it is possible to write a query that counts the number of unfulfilled orders for each part. And if the company wants to mail out expensive catalogs to customers who have ordered more than a thousand dollars of merchandise in the past year, a query can be used to generate the names and addresses of those customers.

A database has a data dictionary that describes the contents of the database. This makes the database self-describing and makes it possible for a user of a program to determine, for instance, the names of the fields available for each record and the relationships between fields of different types of records. The schema is defined and modified by use of a data definition language (DDL). It is important to distinguish between the schema and the actual data. The actual data are sometimes termed the database instance for clarity. The database schema rarely changes once the database has been designed, while the database instance is typically modified quite frequently. The schema is created using the DDL and the data are loaded, queried, or modified using the DML.

Different classes of users have different perspectives on the logical structure of the data. A database management system permits the definition of many views of the data. A view can be a subset of the database or can contain data derived from the database. For example, a view can be defined that contains the total number of unfulfilled orders for each part, which is the information needed by a parts supplier.

A database can be used to enforce business rules. For example, if the company wants to ensure that a catalog is mailed to every new customer, this can be done by having the new-customer application program send a message to the person responsible for mailing catalogs.

Databases also make it possible to make the data secure. For example, MOCK may want to allow its suppliers to query the database to check how many new orders have been placed for their parts. However, the suppliers should not be allowed to access the names and addresses of the customers themselves. A database system makes it possible to set up schemes in which users are denied or allowed access to parts of the database.

Another advantage of using a database system is that the chances of business data being lost are reduced. Databases implement sophisticated recovery schemes that make them immune from many kinds of computer failures. Further, the database can be copied and stored at another location to ensure against the computer system being destroyed by a fire or flood.

Use of a database system requires initial investment in setting up the database system and training personnel in its use. A database system requires hardware and software purchases, the data must be organized so that applications can

conveniently access the database, and application programs need to be written. Also, there are maintenance costs in keeping the computer system running and tuning the system when the size of the database or the workload changes.

Actors in a Database Environment

The set up and use of a database system requires many kinds of personnel in addition to the actual users.

Database designers first determine the needs of all potential users of the database. They then determine what (and how) data are to be represented in the database. The database schema can then be defined using the DDL. Often, the schema is separated into logical and physical schemas. The logical schema defines the structure of the database while the physical schema defines the storage structures for the data. For example, the database designer may decide that the database needs to contain a table of customers and each customer record must contain the fields name, address, and phone number with name and address stored as variable length strings and the phone number as a 10-digit integer. In determining the physical storage, the database designer may decide to build an index on the name field to permit quick access for applications that retrieve customer records by specifying the name field. The database designer may also construct logical views of the database that permit classes of users to see the data they need in the form they want.

System analysts and application programmers develop the applications that will be run against the database. The system analyst develops specifications of the applications and the application programmer implements the specification as a program.

Once the database has been set up, the database administrator (DBA) is responsible for day-to-day operations. The DBA is responsible for authorizing access, monitoring use, acquiring additional hardware or software, tuning the database system, and fixing any problems that arise. The DBA may be assisted by a staff that includes operators and maintenance personnel.

DBMSs and Data Models

A DBMS implements a data model that defines how data will be represented and manipulated. A data model defines a language for representing data and the relationships between data (the DDL) and a language for performing operations against data (the DML). Here, we give an overview of the various data models. The following section 2 provides a detailed discussion. Data models can be compared along some important dimensions. A data model is either value oriented or object oriented. In object-oriented models, it is possible for one object to have a reference to another object. Value-oriented models permit references from one record to another only through common values and are considered to be more amenable to automated optimization of data access. Data models differ in the mechanisms they provide to deal with redundancy. Object-oriented models permit sharing of a single copy while value-oriented models bank on appropriate database design. Data models have differing ways of modeling many-to-many relationships. An example of a many-to-many relationship is that a part has many suppliers and a supplier supplies many parts.

Early DBMSs were based on hierarchical models in which the schema consisted of record types organized in hierarchies by means of links. For example each order record could be linked to the record of the customer who placed the order. The hierarchical model naturally represents such one-to-many relationships (many orders for each customer but not vice versa). Many-to-many relationships can only be represented indirectly. The network model is a generalization in which the links are not restricted to be a hierarchy. In other words, the network model allows the representation of multiple one-to-many relationships for the same member record type as well as a direct representation of many-to-many relationships. Both hierarchical and network models provide a navigational language that can be embedded in application programs written in programming languages such as COBOL.

The relational model represents a database as a collection of relations (i.e., tables). Each table consists of a collection of rows, each of which represents a record. Relationships between records in different tables are represented by storing matching values in the records. For example, customers may be assigned a unique customer key that may be stored as a field of the customer record (row). An order record may then have a custkey field in which the key of the customer placing the order may be stored. This method of modeling permits not just one-to-many but many-to-many relationships to be modeled. The relational model also offers a powerful DML that permits sophisticated questions. A major advantage of the relational model is the use of a declarative DML. In other words, the DML permits a user to specify what operations need to be performed against the data without specifying how they will be done. The DBMS takes on the responsibility of translating the user request into an efficient method of performing the operations. This increases the productivity of the application programmer. Further, application programs are data independent. In other words, the physical storage of the data may be changed without requiring modification of the application programs.

The object-oriented data models combine facilities offered by object-oriented programming languages with database concepts. They offer features such as complex object types, classes, encapsulation, inheritance hierarchies for classes and types, and object identity. For example, the schema may define classes such as customers and orders. Operations such as creating a new customer or adding an order for a customer will be defined as operations (called methods) as part of defining the classes rather than as application programs built on top of the database. Further, the classes are encapsulated in the sense that only the defined operations may be performed against the data. The customer class may also be specialized into subclasses such as individual_customers and corporate_customers. An important feature of an object-oriented data model is object identity. The system is responsible for generating and maintaining identifiers that can be used to reference objects.

Object-relational models combine some features of object-oriented data models with the relational model. Such systems provide features such as the ability to add new data types to a system as well as define complex types using the base types as components. For instance, maps may be added as a new data type along with functions such as finding the shortest distance between two points. Complex types such as sets, lists, and arrays may also be created. Types may also be cre-

ated as subtypes of existing types thus forming an inheritance hierarchy. An object-relational system may also provide a rule system in which condition-action rules get triggered by actions such as update, insertion or deletion of objects (1).

There is considerable interest in searching semistructured data such as that made available by the emergence of the World Wide Web. A common model is to treat a web page as a sequence of words. A query consists of desired combinations of words.The answer to the query is a set of web pages ranked by how closely they match the desired combination of words. Semistructured data models are also emerging that provide more sophisticated ways of modeling and querying data that do not have a regular structure.

Advanced Facilities in a DBMS

Some DBMSs offer advanced facilities such as high availability, parallel execution, data distribution, and gateways.

High availability means that the database system has a low failure rate. The availability of a system may be defined as the fraction of the offered load that is processed with acceptable resonse time. A system is considered well managed if it is available 99.9% of the time or, in other words, has no more than 526 minutes of downtime per year. It is considered fault-tolerant when the availability reaches 99.99% and highly available at 99.999% (2).

A parallel database system has the ability to exploit multiprocessor computers to deliver higher performance. A shared-memory multiprocessor (SMP) has several central processing units (CPU) with a shared memory. A cluster consists of many SMPs connected by a high-speed interconnect. Parallel database systems implement special techniques such as partitioning the data as well as the operations among processors in order to get the work done faster.

A distributed database permits data to be stored on several computers connected by a network. Such data management is useful for enterprises that are geographically distributed. For instance, customer data may be partitioned between the New York and San Francisco sales offices of a company. It is also possible for data to be replicated. Data replication may be synchronous or asynchronous. With synchronous replication, all copies of data are kept exactly synchronized and consistent. If any copy is updated, the DBMS immediately applies the update to all other copies within the same transaction. With asynchronous replication, copies or replicates of data will become temporarily out of sync with each other. If one copy is updated, the change will be propagated and applied to the other copies as a second step, within separate transactions, that may occur with a time delay.

Gateways are a layer of software that emulate the interface for a specific DBMS on top of another DBMS, thus making it possible for tools or applications developed for one DBMS to work with the other DBMS.

Database Standards

Standardization of database languages makes DBMS products interchangeable. It reduces the costs of training personnel and porting applications. Database management system products that support the same standard interfaces may still differ in implementation characteristics such as performance, reliability, and availability, thus giving customers the ability to choose the DBMS that best meets their needs. In practice, DBMS products support the standard interfaces but also provide nonstandard extensions.

Standards may be created by national bodies such as American National Standards Institute (ANSI), international bodies such as International Organization for Standardization (ISO), or industry consortia. A de facto standard may also emerge if a specific product dominates the marketplace forcing other vendors to conform to the interfaces defined by the dominant product.

The Conference on Data Systems Languages (CODASYL) set up a Data Base Task Group (DBTG), which defined standards for the network data model. The X3H2 committee of ANSI has also proposed a standard network language called Network Definition Language (NDL).

The Structured Query Language (SQL) has been standardized by ANSI and ISO. The X3H2 committee of ANSI produced the SQL86 standard based on IBM's implementation of SQL in 1986. This was accepted by ISO as international standard in 1987. An extended standard, SQL-89, was produced in 1989 and SQL2 (also called SQL-92) in 1992. Versions of SQL have also been adopted as standards by X/OPEN and FIPS.

The SQL3 standard, an extension of SQL2, is expected to standardize object-relational systems and is currently being developed by ANSI. The ODMG-93 standard for object-oriented databases was developed by members of the Object Database Management Group (ODMG), a consortium of object-oriented database companies.

Database Market Place

Setting up a database system requires the purchase of several pieces of hardware and software and expertise to put all the pieces together and to write any custom software (such as application programs). One approach to setting up a database system is one-stop shopping. A single vendor supplies all the needed components, puts them together, and makes the system operational. A different approach is a mix and match approach in which components are independently purchased and then integrated to form a full system.

The one-stop shopping approach has the advantage of reducing the risk of the system not working as expected. The single vendor can be held responsible for any problems. Future maintenance and enhancements can come from the same vendor. One-stop shopping results in simplified decision making and is attractive to companies that desire low technology risks.

The disadvantage of one-stop shopping is that the customer gets only the technology that the vendor is willing to supply at the price set by the vendor. Further, once the initial investment has been made, the customer can be locked into proprietary technology from the vendor and cannot benefit from new technologies or lower pricing from other vendors.

The mix and match or open systems approach makes it possible to choose each component independently based on the best match for the need at hand. This often results in technically superior solutions or reduced costs. However, it requires the customer either to take on the responsibility of integrating all the chosen components or obtain the services of a system integrator. It also makes it difficult to troubleshoot problems and to maintain and enhance the system.

Database Applications

Database applications may be classified into multiple categories based on the kinds of operations performed on the data. Applications are also classified based on the business area they model, or on the architecture used in constructing the application.

Application Classification Based on Workload. On-line transaction processing (OLTP) applications typically retrieve, update, insert, or delete single records. Examples are banking transactions such as depositing or withdrawing money, charging a purchase to a credit card, or making an airline reservation. While individual requests are quite simple, an OLTP system must be able to support a large number of concurrent users while providing low response times. It must also ensure that the data remain safe when the computer system fails and that each user gets a consistent view of the data they access.

Data are typically collected in a database by OLTP applications and analyzed by decision-support system (DSS) applications. These applications pose complex queries that require scanning large portions of the database. For example, a query might find the average account balance for customers of different age groups. Two important classes of DSS applications are data mining and on-line analytic processing (OLAP).

Data mining deals with methods for finding trends or patterns in data. For example, a store may want to determine which products are commonly purchased together. This information may be useful in determining how to place products on shelves or develop promotional programs.

On-line analytic processing applications provide a business-oriented view of the data. Rather than deal with data as consisting of tables with rows and columns, OLAP tools present a multidimensional view of data. For example, sales data may be viewed as total sales for a product for each geographical region for each time period. Product, region and time period are dimensions on which sales data may be viewed. For instance, time period may be considered at the granularity of years, quarters, months, or weeks, thus yielding a hierarchy. So sales data may be subjected to queries such as "find the total sales for all products for each quarter in the northern sales region."

Benchmarks. A benchmark consists of a workload and a set of metrics. It is used for quantitative comparison of alternate configurations of hardware and software. As an example, the TPC-C benchmark, defined by Transaction Processing Council (TPC), models OLTP applications. The workload consists of five types of transactions that might be run by a wholesale supplier using a database to manage orders. The benchmark produces two metrics, tpmC, which measures performance as the number of transactions the system can run per minute and price/performance as $/tpmC.

Common uses of benchmarks are to compare competing DBMSs on the same hardware, competing hardware for the same DBMS, and new releases of a DBMS with the old product.

A good benchmark must be relevant to the application in the sense that the workload should represent the typical operations and the metrics should be meaningful measures of performance and price/performance. The benchmark must be understandable to people with a nontechnical background for it to gain credibility. Finally, the benchmark should be designed so that it can be run on many different systems and architectures and should apply to small and large systems.

The Transaction Processing Council is a consortium of vendors that defines database benchmarks and standard ways for measuring and reporting results. It also defines the process for certifying a result and sets guidelines for how the results may be used. For more information on benchmarks the reader is referred to Refs. 3 and 4.

Application Architectures. A typical application may be regarded as consisting of three components: presentation, application logic, and database. The presentation refers to the user interface and the application logic refers to the tasks and rules that implement the needs of the business. Depending on how well the software is separated into the three components, application architectures may be broadly classified as monolithic, two tier and three tier.

Early database applications were built for mainframe computers. Users typically had dumb terminals on their desks. Terminals were connected to a central mainframe computer and communication between the computer and terminal was character based. Applications were monolithic and resided entirely on the mainframe.

In the 1980s, distributed computing became popular and the terminal was replaced by desktop computers that had graphics and could run programs. This has led to architecture of applications with two tiers with the presentation layer sitting on the desktop (the client) and the database running on a separate shared server. The application logic can either be part of the client or the server, yielding the fat-client and fat-server variants of the two-tier architecture. Since a server caters to the needs of a large number of clients, the fat-client architecture has the advantage of reducing the load on the server. However, it requires one copy of the application logic to be placed at each client. This is problematic from the perspective of security, availability, and system maintenance. For instance, all machines may need to be upgraded simultaneously when a new version of the application becomes available. The fat-server architecture places the application logic with the database server. This logic may either work on top of the database or reside inside the database as stored procedures.

The 1990s have seen the development of three-tier applications in which all three components are clearly separated and may be put on different machines. Databases reside on the bottom tier on powerful server machines such as mainframes and high-end workstations. The middle tier consists of workstations and hosts the application logic; it may also include the consolidation of data from multiple databases into a data warehouse. The top tier consists of the presentation services and usually runs on personal computers.

Packaged Applications. Building and maintaining sophisticated applications is sometimes regarded as an expensive and risky undertaking. This has motivated many companies to buy packaged applications rather than build custom applications. The supplier of the packaged application takes on the responsibility of maintaining the application and of enhancing it as the needs of the business change. Since a packaged application must cater to the needs of a wide variety of com-

panies, these packages are built to be flexible, which requires extensive customization of the package before it can be put into operation. An example of a packaged application is a human-resources package, which might provide functionality such as managing resumes of applicants, salary and benefits for employees, and pensions for retirees. Packages may also ensure compliance with the law and incorporate taxation rules.

DATA MODELS

At the early development of database systems, it was almost axiomatic that there were three important data models: hierarchical, network, and relational. This view is slowly losing ground as the relational model becomes the most popular data model and other new semantic data models emerge. With this perspective, we will present the relational data model in some depth and provide only brief overviews of the network and hierarchical models, which are mainly of historical importance. Object-oriented and object-relational systems will be discussed in somewhat more detail.

Relational Data Model

The relational data model was proposed by E. F. Codd (5) in 1970. He also introduced relational algebra and relational calculus as the mathematical foundation for manipulating data stored in relations. Codd received the 1981 ACM Turing Award for his work on the relational data model. The primary reasons for the popularity of the relational model are its presentation of data in familiar tabular form and its powerful declarative data manipulation language. The relational data model is based on a simple and uniform data construct known as a relation. The results of operations defined on relations are themselves relations; thus, these operations can be combined and cascaded easily. As shown in Fig. 2, a relation can be viewed in a tabular form where a row represents a collection of related values of a real-world entity.

Basic Concepts. The mathematical concept behind the relational model is the set-theoretic ***relation,*** which is a subset of the Cartesian product of a list of domains. A ***domain*** D is set of atomic values; the requirement that elements of domains be atomic means that they are not divisible into components within the framework of a relational data model. The atomicity of domains leads to *first normal* form of the classical relational data model. A ***relation schema*** $R(A_1, A_2, \ldots, A_n)$ contains a relation name R and the list of attributes, $A_1, A_2, \ldots, A_n$; each attribute name A_i, is defined over some domain D_i. A relation schema describes a relation. The degree of a relation is the number of attributes n in its schema. The relation R, therefore, is a set of n-tuples; each tuple is an ordered list of n-values $\langle v_1, v_2, \ldots, v_n \rangle$, where $v_i (1 \leq i \leq n)$ is an element from domain D_i of attribute A_i. A relation is a set; hence its elements (i.e., tuples) are distinct and have no inherent ordering associated with them.

A ***key*** (also called unique key) of a relation is the minimal subset, which is not necessarily proper, of attributes of the relation schema such that no two tuples in the relation contain the same combination of values for these attributes; a key value therefore uniquely identifies a tuple. Note that the key is determined from the semantics of key attributes, not from its current values in the relation. A relation may, in general, have several keys, one of which is designated as the ***primary key.*** We use the convention that attributes that form the primary key of a relation schema are printed in boldface, as shown in Fig. 3.

Entity integrity constraints states that no primary key value can be fully or partially null; null is a special value that implies missing or unavailable information. Since primary key values are used to identify a tuple in a relation, A null value cannot be allowed. a set of attributes in relation R1 is said to satisfy the ***referential integrity*** constraints with respect to relation R2 (R1 and R2 are not necessarily distinct), if the following hold:

1. the attributes in R1 have the same domains as the primary key in R2;
2. the values of the attributes in a tuple in R1 either have the same values as the primary key in some tuple in R2 or are null.

The set of attributes in R1 is called a ***foreign key,*** which is said to *reference* the primary key in R2. The foreign key reference between relations represents a relationship between real-world entities. Note that in the relational model both entities and relationships are represented by relations.

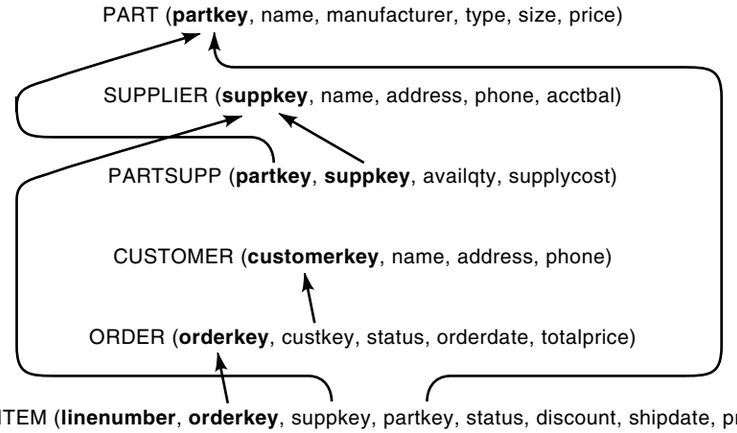


Figure 2. The database schema.

PART

partkey	name	manufacturer	type	size	price
P1	bolt	anderson	copper	7	0.45
P2	nut	universal	anodized	9	1.21
P4	screw	clark	burnished	11	1.11
P5	cog	universal	plated	5	5.35

SUPPLIER

suppkey	name	address	phone	acctbal
S12	Jackson	11 Main St, S.F.	4155551212	900.00
S13	Onan	10 3rd Ave, S.J.	4085554321	896.98
S14	Levine	NULL	2125554379	789.11
S15	Smith	9 55th St., N.Y.	NULL	55.12
S16	Chen	NULL	5107773412	127.87

PARTSUPP

partkey	**suppkey**	availqty	supplycost
P1	S12	100	12.85
P1	S13	85	25.64
P2	S12	65	12.89
P2	S14	90	15.00
P5	S15	110	13.99

Figure 3. Relational database content.

Relation schemas are shown in Fig. 3. In relation `PART`, `partkey` is the primary key. The primary key for relation `PARTSUPP` is the combination of attributes `suppkey` and `partkey`. The domains of the attributes, `partkey` and `size`, are character string and integer respectively. The degree of the relation `PART` is 6. The arrows in Fig. 3 represent foreign key to primary key references; for example, `partkey` in `PARTSUPP` is a foreign key that references the primary key `partkey` in `PART`. The relation `PARTSUPP` represents a relationship—which supplier supplies which parts—between parts and suppliers indicated by `PARTSUPP`'s foreign key references to `PART` and `SUPPLIER`.

An example of relation `PART` is shown in Fig. 2. Each tuple in relation `PART` corresponds to a particular part in the real world. The various attribute values in a tuple describe that part. A tuple in `PART` is <P2, nut, universal, anodized, 9, 1.21>, where "P2" is the partkey of the part and "nut" is the name of the part, and so on. Similarly, relation `SUPPLIER` represents information about suppliers. A tuple in relation `PARTSUPP`, <P1, S12, 100, 12.85>, indicates that supplier S12 supplies part P2 in quantity 100 and the cost of this shipment is $12.85.

Data Manipulation in the Relational Model. The DMLs, relational algebra and relational calculus, provide the theoretical basis for expressing operations on relation. In *relational algebra,* specialized algebraic operators are applied to relations in order to express queries. In *relational calculus,* queries are expressed by writing logical formulae that the tuples in the result must satisfy. Relational algebra and relational calculus can be shown to be equivalent in their expressive powers. Any relational data manipulation language that has as much expressive power as relational algebra (or relational calculus) is called a *relationally complete* language. A declarative (i.e., nonprocedural) query language allows users to describe what they want without having to specify the procedure for retrieving the result. Relational calculus is considered somewhat more declarative than relational algebra.

Relational Algebra. Relational algebra has five primitive operations: *union* ($\cup$), *set difference* ($-$), *Cartesian product* ($\times$), *projection* (π), and *selection* (σ). There are three additional nonprimitive operations—*intersection, join,* and *division*—that are defined in terms of the primitive operators. The operands of relational algebra are relations; the result of these operations is also a relation; this is called the closure property of relational algebra. The closure property facilitates composition of a sequence of operations. Operations such as union, set difference, and Cartesian product originate from the set theory; the others have been devised specifically for the relational model. We will not define all the operations here, but present a brief sketch of a few of them. A detailed discussion of relational algebra operations can be found in Ullman (6).

The selection operation retrieves a subset of tuples from a relation, which satisfies a given predicate; the selection symbol, σ, is followed by a Boolean expression. The projection operation chooses specified attributes from a relation and discards the remaining attributes; the projection symbol, π, is followed by a list of attributes. The Cartesian product operation combines two relations by concatenating each tuple from one relation with every tuple in the other relation. The join operation is defined in terms of a Cartesian product of two relations followed by a selection predicate on the resulting relation. Thus the join operation combines two relations on the values of some of their attributes. A query is expressed as a sequence of relational algebra operations. The sequence of operations in relational algebra seems to specify a partial strategy for evaluating the query.

Consider a query that retrieves the name and type of the parts that are supplied by the supplier whose suppkey is S12. This information comes from relations `PART` and `PARTSUPP`; the attribute that is used for joining them is partkey in both relations. This query can be expressed in relational algebra as the following:

$$\pi_{\texttt{name, type}}\{\sigma_{\texttt{suppkey='S12'}}[\sigma_{\texttt{PART.partkey=PARTSUPP.partkey}}(\texttt{PART} \times \texttt{PARTSUPP})]\}$$

Relational Calculus. In relational calculus, we write declarative expressions to specify the query. Relational calculus is a formal query language based on the branch of mathematical logic called first-order predicate calculus. There are two ways in which the predicate calculus can be applied to relational data manipulation language. These are called ***tuple relational calculus*** and ***domain relational calculus.*** The difference between the two is that in tuple relational calculus, variables in the formulae range over tuples in a relation. In domain relational calculus, variables range over domains of attributes. The formulation of the above query in tuple relational calculus takes the following form.

```
{X.name,X.type | PART(X) ^ [(∃Y)PARTSUPP(Y) ^
   X.partkey = Y.partkey ^ Y.suppkey = 'S12']} |
```

Relational query languages such as SQL and QUEL are essentially based on tuple relational calculus. A graphic query

language called Query By Example (QBE) borrows its basic notions from domain relational calculus. For further information on this topic, see Ullman (6) and Maier (7).

Query Language SQL. Structured Query Language (SQL), formerly known as SEQUEL, was developed by IBM for an experimental relational database system called System R. It is now the most commonly used query language for commercial relational database systems (8,9). As a declarative query language, SQL provides a syntactical sugaring of the tuple relational calculus. SQL contains statements for query, update, and data definition; that is, it is both a DDL and a DML.

SQL uses the term table, which is similar to relation; the difference is that a table in SQL permits duplicate rows; a tuple is also called row and an attribute is called column.

The basic SQL queries are a select statement of the form:

```
SELECT Rj,A1, ..., Rk.Ar
FROM R1, ..., Rn
WHERE <predicate>
```

`SELECT, FROM,` and `WHERE` are SQL keywords. Here, R1, . . ., Rn is a list of relations (tables), which forms the from clause Rj.A1, . . ., Rk.Ar, the select clause, is a list of attributes (columns). The qualified attribute of the form R.A refers to the attribute A of relation R; it is used to distinguish between attributes of the same name in different relations. The relations in the select clause are a subset of the relations listed in the from clause. The <predicate> is a Boolean expression involving logical connectives conjunction (and), disjunction (or), and negation (not) and comparison operators $=$, $\leq$, $\neq$, etc., and qualified relational attributes. The <predicate> specifies a selection condition (i.e., Boolean expression) for tuples to be retrieved. There is a notational conflict between relational algebra and SQL; the keyword SELECT in SQL corresponds to projection (π) in relational algebra, not to selection (σ).

The execution semantics of an SQL query is the following:

1. take the Cartesian product of all relations specified in the from clause;
2. apply the restriction predicate specified in the where clause on the resulting relation; and
3. project out the attributes specified in the select clause.

A query execution may not always follow this sequence but it must produce a result that is equivalent to the one given by the three-step method described before.

We illustrate the basic select statement of SQL.

Q1. Retrieve the name and type of the parts that are supplied by the supplier whose suppkey is S12.

```
Q1: SELECT PART.name, PART.type
    FROM PART, PARTSUPP
    WHERE PART.partkey = PARTSUPP.partkey AND
    PARTSUPP.suppkey = 'S12'
```

The query Q1 shows a retrieval based on the join of two tables. This is the same query which was expressed using relational algebra in the previous section. The predicate, `PART.partkey = PARTSUPP.partkey`, specifies an equality join (also called equi-join) between the two relations; the columns in this predicate are called join columns.

Sometimes a query needs to refer to two or more tuples in the same relation. This is achieved by defining several tuple variables for that relation in the from clause and using the tuple variables as aliases of the relation in the rest of the query.

Q2. Retrieve the name and partkey of parts which are priced higher than the part with partkey P2.

```
Q2: SELECT Y.name, Y.partkey
    FROM PART X,  PART Y
    WHERE X.partkey = 'P2' AND
    Y.price > X.price
```

The from clause of query Q2 shows the SQL syntax for declaring aliases. Here X and Y both are aliases of `PART`, in effect making them tuple variables that range over *different* instances of relation `PART`.

SQL is a relationally complete language; hence, it provides language constructs that are equivalent to all the relational algebra operations. Consider the following query.

Q3. Retrieve the name of suppliers who do not supply plated parts.

```
Q3: SELECT SUPPLIER.name
    FROM SUPPLIER
    WHERE SUPPLIER.suppkey NOT IN
    (SELECT PARTSUPP.suppkey
    FROM  PART, PARTSUPP
    WHERE PART.partkey = PARTSUPP.partkey
        AND PART.type = 'plated')
```

The query first finds all suppliers who supply plated parts; it then uses the set difference operation to discard the suppliers found in the first step from a list of all suppliers, thus, in effect, selecting the suppliers who do *not* supply plated parts. The finding of the first category of suppliers is done by the nested subquery in Q3; the set difference operation is achieved by using the SQL comparison operator ``NOT IN''.

SQL provides more expressive power than relational algebra or tuple calculus by providing aggregate functions and sorting of results. The aggregate functions, unlike other SQL operations, do not apply to one tuple at a time but to a collection of tuples that are returned by the query. There are five standard aggregate functions: `SUM, COUNT, MAX, MIN,` and `AVERAGE`.

SQL also provides a language feature called `GROUP BY`, which partitions the tuples of a relation into groups; an aggregate function then applies to the groups individually. Aggregate functions can be used without the group by clause; in this case, the aggregation applies to all the tuples returned by the query. If only a subset of the groups is relevant, then a having clause can be used to filter out the unwanted groups formed by the group by clause. This filtering is independent of any filtering specified in the where clause that applies to tuples in a relation and is done *before* the grouping takes place.

Q4. Find the partkey and the average supply cost of parts whose average supply cost exceeds $30.00.

```
Q4: SELECT PARTSUPP.partkey,
            AVG (PARTSUPP.supplycost)
                FROM  PARTSUPP
    GROUP BY PARTSUPP.partkey
    HAVING AVG (PARTSUPP.supplycost) > 30.00
```

SQL provides a facility for the definition of views. Views permit the user to perceive the database in terms of just those

derived relations that directly belong to their applications. **Views** are relations that are defined in terms of base relations and previously defined views using the SQL select statement. A view does not necessarily exist in the physical form; hence it is considered a virtual relation in contradistinction to base relations that are actually stored in the database. Consider the following definition of a view that shows the partkey, name, manufacturer, and suppliers of the parts that cost more than $5.00.

```
CREATE VIEW  EXPENSIVE_PART
SELECT  PART.partkey, PART.name,
        PART.manufacturer, PARTSUPP.suppkey
FROM  PART, PARTSUPP
WHERE PART.price > 5.00
      and PART.partkey = PARTSUPP.partkey
```

The system maintains the name and the definition of views. Any reference to a view name (e.g., `EXPENSIVE_PART`) in a SQL statement is substituted with the definition of the view. This is called view resolution

In the preceding discussion, we have concentrated on SQL queries and view definition. SQL also has syntax for inserting a new tuple, deleting and modifying an existing tuple, but we will not discuss these operations here.

Relational Database Design. The central idea behind the design of relational schemas is that of data dependency of attributes, which means that some attributes uniquely determine other attributes in a relation. We have seen this dependency in the definition of relational key. These dependencies are called *functional dependencies.* Informally, a prescription for a good database design is to disallow all functional dependencies other than key dependencies. Nonkey functional dependencies may lead to serious problems in the database. Consider the following relation schema that contains attributes from `PART` and `PARTSUPP`.

```
PS (partkey, suppkey, supplycost, price,
type)
```

The attributes, `price` and `type`, belong to `PART` and hence are functionally dependent on `partkey`, the primary key of `PART`. It should be emphasized that `partkey` is *not* the primary key of relation `PS`. This dependency generally leads to two classes of problem: redundancy and update anomaly, which we will explain by an illustration. *Redundancy* refers to the fact that values of attributes, `type` and `price`, of a part need to be stored in `PS` as many times as there are suppliers of the part. The other related problem is the potential inconsistency that may ensue if all the instances of a part's price, for example, are not consistently modified in `PS` when a change occurs; thus a part may end up having multiple—that is, inconsistent—prices. The *update anomaly* refers to a situation in which a part and its attributes cannot be recorded in `PS` unless that part is supplied by some supplier,

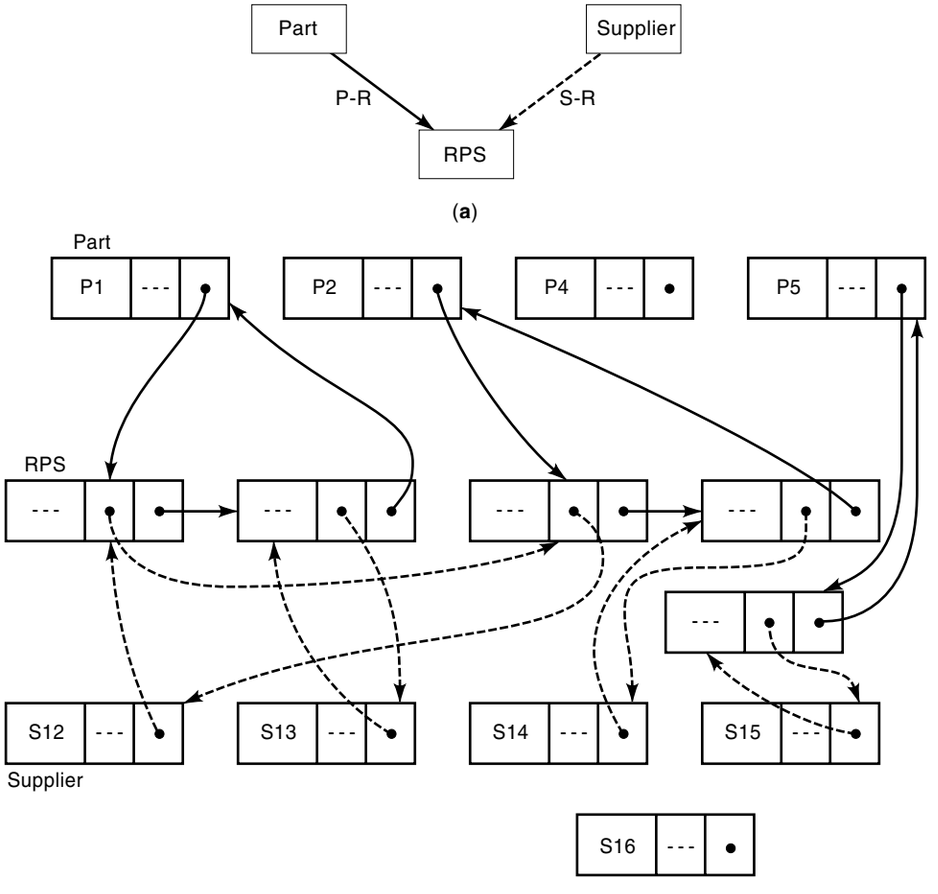


Figure 4. A network database. (a) Schema, (b) content.

because this implies putting a null value in suppkey, which is a subset of the primary key of PS. As mentioned before, the entity integrity rule disallows a primary key value to be fully or partially null. The related inverse problem is that if a part is no longer supplied by any supplier, then we must remove the part and all its information from PS.

A good database design is commonly measured in terms of whether relations in the database are in ***third*** or ***Boyce-Codd normal form.*** The imposition of these normal forms on a relation results in its decomposition into smaller constituent relations that are free from redundancies and anomalies. (There are other normal forms, such as the fourth normal form that ensures that a relation does not have multivalued dependency.) The relation schemas shown in Fig. 3 are in Boyce-Codd normal form (8), and thus they represent a good database design.

Network Data Model

Early work on the network data model was done by C. Bachman during the development of the first commercial network DBMS called IDS; he also proposed a diagrammatic technique for representing relationships in database schemas. Bachman received the 1973 ACM Turing Award for this work. The actual network data model and language constructs were defined by the Conference on Data Systems Language (CODASYL) committee in Database Task Group (DBTG) report in 1971. In the following section, we will highlight the central concepts of the network data model rather than discuss the specific details of DBTG (or CODASYL) data model.

The Network of Records and Links. There are two basic data constructs in the network model: record and link. Data are stored in records as a group of related data values. The ***record*** type describes the structure of a group of records that store the same type of information. The record type and record bear close correspondence with relation schema and tuple of the relational model. The ***link*** type (somewhat inappropriately called set type in the DBTG report) contains a description of a one-to-many relationship between two record types. Each link type contains the name of the link, an owner record type, and a member record type. In Fig. 4(a), P-R is a link type whose owner is the Part record type and whose member is the RPS record type. The link type is represented as a directed edge from the owner to the member. These one-to-many binary relationships form a directed acyclic graph (network) of related records. A link is composed of one owner record and zero or more member records. A member record cannot exist in more than one link of a particular link type; this requirement, in effect, imposes a one-to-many constraint. This constraint does not preclude a member record from participating in multiple links of different link types.

A link in the network data model allows only a binary one-to-many relationship. A many-to-many relationship can be represented by the use of an additional virtual record type and two link types that contain that virtual record type as members. The owners of these two link types are the record types whose many-to-many relationship is being represented. Figure 4(a) shows a network schema that uses RPS record types for representing a many-to-many relationship between record types Part and Supplier. The relationship is the same as shown in relation PARTSUPP of Fig. 3.

Stored Representations of Link. A link is commonly represented in the database as a ring (circularly chained list) linking the record of the owner of a link to that of all its members. As can be seen from Fig. 4(b), every record has one or more labeled pointer fields that are used to chain the records in a link. The RPS records have two different kinds of pointers; the solid and dashed pointers are used to represent respectively P-R links (which chain them with their owner of type Part) and S-R links (which chain them with their owner of type Supplier). A query in the network data model requires the navigation of these chains to find and retrieve one or more related records.

Data Manipulation in the Network Model. The data manipulation language in the network data model is a procedural, record-at-a-time language that requires explicit navigation of the network of chained records by the application program. This should be contrasted with the relational query language SQL, which is a set-at-a-time language that allows formulation of queries in a declarative manner. We will illustrate this point by an example.

Consider query Q2 discussed before. In the network model, in order to retrieve all parts that are supplied by the supplier S12, we start with the particular supplier record, then navigate through all the RPS records that the supplier S12 owns using the next pointer of S-R type shown as dashed arrows in Fig. 4(b). For each record, we determine the owner of the record using the P-R pointers shown as solid arrows. A fragment of code in a pseudo programming language shows the query formulation in the network data model.

```
Supplier.suppkey = 'S12'
FIND ANY Supplier using suppkey
If Found Then
Begin
    FIND FIRST  RPS WITHIN  S-R
    While Found Do
    Begin
        GET RPS
        FIND OWNER WITHIN P-R
        GET Part
        print (Part.name, Part.type)
        FIND  NEXT WITHIN S-R
   End
End
```

The above query uses DBTG commands FIND and GET. There are many variants of FIND, all of which locate the relevant record and mark it as the current record of its link and record types. GET simply retrieves the current record into the application's work-space.

Hierarchical Data Model

A hierarchy is a directed graph that is a forest; that is, a set of trees. The hierarchical database systems were based on hierarchical organizations, taxonomic classification of organisms, or other such hierarchical classifications that are popular in the real world. Hierarchical database systems, however, were not constructed on the basis of a predefined data model; on the contrary, such a model was defined after the event by a process of abstraction from the implemented system. This might provide us with some insight as to why the hierarchical

data model is relatively ill defined. The hierarchical data model is capable of representing hierarchical structures in a direct and precise way. However, it proves to be quite inadequate in representing nonhierarchical structures such as many-to-many and *n*-ary relationships.

IBM's Information Managment System (IMS) and SAS Institute's System-2000 are two well-known commercial hierarchical database systems. The first version of IMS was released in 1969, and was one of the earliest commercial database systems. In the mainframe market place, IMS continues to be one of the most widely used products, although this may not remain true for long.

Hierarchical Data Structure and Manipulation Language. A hierarchical database consists of an ordered set of multiple occurrences of a single type of tree. The hierarchy or tree contains a number of parent-child relationships (PCR), which are asymmetric and one-to-many. As mentioned before, a strict hierarchical model cannot represent many-to-many or *n*-ary relationships, nor can it represent the case where a record may have to participate as a child in more than one PCR. A notion of virtual (called "logical" in IMS) record or pointer is employed to deal with these problems; a record can participate in two PCRs, if one of the two parents is virtual. The introduction of virtual records effectively transforms the hierarchy (tree) into a network (directed graph). The data manipulation language of hierarchical database is a record-at-a time language, which requires explicit navigation of hierarchical occurrences in the database. In this model, the formulation of nonsimple queries becomes a more cumbersome task than that of the sample shown for the network data model.

Object-Oriented Data Models

In recent years, object-oriented technology has achieved wide acceptance, maturity, and market presence. It is the next generation for application development. This new paradigm has significantly improved the programmer's productivity and lowered the cost of application development. Object-oriented database systems (OODS) were introduced in the late 1980s to meet the needs of emerging complex applications and to deal with some of the inherent limitations of the relational model. They were proposed partly in response to the anticipated growth of the use of object-oriented programming languages (OOPLs). Object-oriented database systems borrowed their paradigm from object-oriented programming languages such as Simula and Smalltalk, which are generally considered to be the precursors of the early OODSs.

We present a brief overview of the key features of object-oriented programming languages that have generally been adopted by OODSs. In OOPLs, the notion of *abstract data type,* called *class,* conceals the internal data structure and provides all possible external operations on the objects of the class; this is known as *encapsulation.* Objects are instances of a class; these objects exist only during the execution of the program. Another key idea of OOPLs is *class hierarchy* and *inheritance;* this allows specification of new classes that inherit much of their structures and operations from previously defined classes, called superclasses. The operations in OOPLs are called *methods.* A related concept is of method *polymorphism,* which refers to the fact that a method name may apply to objects of different classes; in such cases, the methods may have different implementations and different semantics. Polymorphism may also require the use of *dynamic binding* of the method name to the appropriate method implementation at run time, when the class of the object to which the method is applied becomes known.

It should be emphasized that there is no agreed upon definition of an object-oriented data model as there is for relational model. We enumerate the features that ought to be supported by object-oriented database systems: *object identifier* (OID), which is a unique system-generated identifier for each object in the system; class references or relationships; complex objects of arbitrary structure and their constructors; encapsulation; class hierarchy and inheritance; and polymorphism. A sophisticated database system must also provide access methods, a powerful declarative query language, transaction management, concurrency control, and recovery (10).

One class of OODSs ties itself closely with an OOPL. These OODSs generally provide a query language, but both the OOPL and the query language execute in the application program environment, sharing the same type system, data structures, and work-space; they can, with some justification, be looked upon as persistent storage managers for OOPL objects. Nevertheless, these systems treat persistent data differently from transient data. One of the perceived benefits of these types of systems is a seamless interface between an OOPL and a database system; that is, an OOPL user will not need to learn a separate database DDL and DML. As long as persistent storage management is the only objective of such a system, the benefit is more or less achievable. However, if most of the database features that have been incorporated into relational database systems are needed in the application, seamlessness is no longer feasible. These OODSs often lack the capability of a powerful declarative query language, metadata management, views, and authorization, although there is a trend toward incorporating some of these features into the system. The lack of well-defined operations in these models led Codd (11) to compare them with organisms that possess "anatomy without physiology." Examples of this type of OODSs include O2, ObjectStore, ONTOS, and VERSANT, which integrate themselves with C++, and GemStone, which uses Smalltalk.

The other class of OODSs extends underlying functional or relational systems with object-oriented capabilities and provide their own SQL-like nonprocedural query language. ADAPLEX, Informix, OpenODB, Orion, Postgres, PROBE, Starburst, and UniSQL are some examples of OODSs of this type (12). Unified relational and object systems (1) extend the relational model with key object-oriented features. They enable users to store their object-oriented application data in databases without compromising the essential features of the relational database that they already rely upon; such features include robustness, high performance, standards compliance, authorization, metadata management, view definition, support for open systems, security, and concurrency control.

The ANSI SQL-3 standards committee is working on the extension of SQL-2 with object-oriented features. There appears to be a consensus that the next-generation database systems will incorporate key relational and object-oriented features with support for management of spatial-temporal, multimedia and active data, and long-duration transactions.

TRANSACTION AND CONCURRENCY CONTROL

There are many applications in which multiple programs need to run concurrently. An example is an airline reserva-

tion system, where several agents may make reservations at the same time, and therefore concurrently change and access the airline database. The canonical problem is that two or more programs accessing the database might reserve the same seat for different persons, if the database management system does not control access to the database.

Multiprogramming allows the computer to process several programs in a concurrent manner. Concurrent programs, by sharing the processor among them, improve the efficiency of a computer system. Even if a computer system comprises only a single central processing unit (CPU), many programs may be processed concurrently by use of multiprogramming; the processor executes some commands of a program, then suspends this program and executes some commands of another program; program execution is resumed at the point where it was suspended when it gets its turn with the CPU. Therefore, concurrent programs are actually *interleaved.* If the computer system has multiple CPUs, *parallel* rather than interleaved execution of a program is possible. Most of the theory of database concurrency control is developed in terms of *interleaved concurrency,* which, in principle, can be applied to parallel concurrency.

Basic Concepts

A ***transaction*** is a single execution of a program that changes or accesses a database. This program may be a simple query or update expressed in database query language or a complex host language program with embedded calls to the query language. The acid test for a transaction's correctness is that it possesses the atomicity, consistency, isolation, and durability (ACID) properties.

Atomicity. The changes made to the state of the database by a transaction are a unit of work; that is, either all happen or none happens. For a transaction to be atomic, it must behave atomically to an outside observer. A failed or aborted transaction has no effect on the state of the database.

Consistency. A transaction transforms a database from one consistent state to another. The actions of the transaction should not violate any of the integrity constraints associated with the state. This requires that a transaction must represent a correct program.

Isolation. Although transactions execute concurrently, each transaction T is isolated from the state changes of other transactions in the sense that other transactions appear to T as if they were either executed before or after T. In other words, the execution of a transaction must take place as it would in a single-user environment.

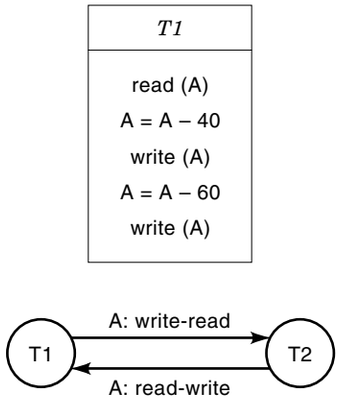


Figure 5. Transaction T1.

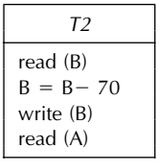


Figure 6. Transaction T2.

Durability. The changes to the state of the database made by a successfully completed transaction survive failures.

A transaction has a well-defined boundary marked by the application at its beginning and end. A transaction that successfully completes is said to have ***committed***. A committed transaction cannot be revoked. Changes made to the database by a committed transaction are durable and can be seen by an outside observer. A transaction might not successfully complete, but might have to ***abort*** or ***rollback***. For example, a transaction may abort because it performed an illegal computation, or it tried to make a change to the database that violated its integrity constraints. An aborted transaction does not change the state of the database in any way; the changes that it might have made to the database are concealed from the view of an outsider observer, and must be undone.

To manage concurrency, the database must be conceptually partitioned into uniquely named database items, the units of data to which access is controlled. Item size is determined by the system and is called granularity. The data operations involved in a transaction can be simplified to the following. T:read (A) and T:write (A). The first means that transaction T reads the data item A into a program variable; the second means that T writes the value of a program variable to data item A; to simplify our notation we will assume that the name of the program variable is the same as that of the data item. Figures 5 and 6 show transactions T1 and T2 respectively. T1 reads and writes data item A; T2 reads and writes data item B and also reads data item A. Figure 7 shows the serial execution of transactions T1 and T2.

Transactional Dependencies and Isolation

Two transactions executing concurrently may have dependencies on each other. There are three types of undesirable dependencies: ***lost update, dirty read,*** and ***unrepeatable read.*** These occur when two concurrent transactions that access or change the same data items have their operations interleaved in such a way that makes some of the transactional operations incorrect. The existence of transactional dependencies implies that there is a violation of isolation. In the follow-

T1	*T2*
read (A)	
A = A − 40	
write (A)	
A = A − 60	
write (A)	
	read (B)
	B = B − 70
	write (B)
	read (A)

Figure 7. Schedule S1: serial.

T3	*T4*
read (A)	
A = A – 100	
	read (A)
	A = A + 50
write (A)	
read (B)	
	write (A)
B = B + 100	
write (B)	

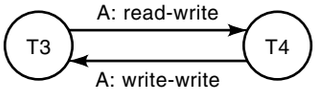


Figure 8. Schedule S2: lost update.

T5	*T4*
read (A)	
	read (A)
	A = A + 50
read (B)	
	write (A)
B = B + 90	
write (B)	
read (A)	

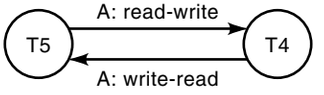


Figure 10. Schedule S4: unrepeatable read.

ing discussion, we treat each of these dependencies in some detail.

Lost Update. This occurs when one transaction's write is overwritten by another transaction, which writes the data based on the original value read. Consider concurrent execution of two transactions T3 and T4, which perform banking applications shown in Fig. 8. T3 shows a fund transfer of amount $100.00 from account A to account B; T4 shows a credit of amount $50.00 to account A. Since T4 overwrites the value of A, the debit to account A is lost in this process, while account B is credited with the transferred amount $100.00. Clearly, this produces incorrect values in the database.

Dirty Read. This happens when one transaction reads a data value previously written by another concurrent transaction, and then the first transaction either rewrites the value or aborts and restores the original value. In Fig. 9, transaction T1 writes A, which is read by T2; however, T1 rewrites the value of A. Thus, the value of A read by T2 is dirty, or incorrect.

Unrepeatable Read. The unrepeatability of read creates inconsistent semantics, as a transaction T must see the same value of a data item on multiple reads provided that the data item is not modified by T. In Fig. 10, T5 reads A again and gets a different value, as it was modified by T4 after T5 read it. Thus T5 has the problem of unrepeatable reads.

The concurrent execution of transactions T1 and T2 shown in Fig. 11 does not have any dependencies and hence is correct. It should be noted that if transactions were not executed concurrently or if they did not change the database, there would be no transactional dependencies.

T1	*T2*
	read (B)
	B = B – 70
	write (B)
read (A)	
A = A – 40	
write (A)	
	read (A)
A = A – 60	
write (A)	

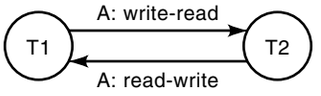


Figure 9. Schedule S3: dirty read.

T1	*T2*
	read (B)
read (A)	
A = A – 40	
	B = B – 70
	write (B)
write (A)	
A = A – 60	
write (A)	
	read (A)

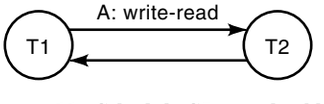


Figure 11. Schedule S5: serializable.

Serializability

We now present a formal discussion of the isolation property. A ***schedule*** (or history) S of transactions T1, T2, . . ., Tn, is an ordering of the operations in these transactions such that all operations in each transaction Ti that participates in S appear in the same order in S as they do in Ti. The operations in participating transactions can, of course, be interleaved in concurrent execution. Figure 7 and Fig. 11 show two possible schedules, S1 and S5, of transactions T1 and T2.

Two operations in a schedule are said to ***conflict,*** if they belong to different transactions, they access the same data item, and at least one of the two operations is a write operation. The notion of conflict implies that the order of such operations is crucial, as the combined effect of the two operations depends upon the order in which they are executed. This leads to three types of conflict: write-write, write-read, and read-write, which give rise to lost update, dirty read, and unrepeatable read, respectively. Interestingly, if these three forms of dependency can be prevented in a schedule, then there will be no concurrency anomalies and the schedule will satisfy the isolation property. An important aspect of concurrency control is the serializability theory of schedules, which attempts to determine whether a given schedule provides isolation.

A schedule S is ***serial,*** if, for every transaction Ti that participates in S, all the operations of Ti are executed consecutively without any interleaving of operations from other participating transactions; that is, in serial schedule, every transaction is performed in a serial order. Figure 7 shows a serial schedule S1, in which transactions T4 follows transaction T3. The schedules shown in Figs. 8–11 are all nonserial. A serial schedule is, by definition, a *correct* schedule because a transaction executed on its own truly satisfies the *isolation* requirement.

A schedule S is ***serializable,*** if it is equivalent to *any* serial schedule of the participating transaction in S. If a nonserial schedule is serializable, then it is equivalent to a serial schedule and thus correct. There are essentially two notions of equivalence: conflict equivalence and view equivalence.

Two schedules are said to be ***conflict equivalent*** if the order of any two *conflicting* operations is the same in the schedules. A schedule Q is said to be ***conflict serializable,*** if it is conflict equivalent to some serial schedule S. In schedule S5 (Fig. 11), the only conflicting operation is T1:write (A) and T2:read (A). S5 is conflict equivalent to the serial schedule S1 (Fig. 7), since the order of the conflicting operation is preserved in S1; hence S5 is conflict serializable, isolated or correct. Note that the order of nonconflicting operations are immaterial.

Test of Conflict Serializability. The equivalence of a given schedule S with n transactions could be determined by comparing S with all possible serial schedules of these transactions; this would be an intractable task, since there are $n!$ possible serial schedules for n transactions. However, there exists a simple algorithm for determining the conflict serializability of a schedule S based on a directed graph approach.

A ***precedence graph*** $G = (N, E)$ consists of a set of nodes $N = \{T1, T2, \ldots, Tn\}$, and a set of directed edges $E = \{e1, e2, \ldots, em\}$. Each transaction Ti in schedule S corresponds to a node in the graph. Each edge in E is an ordered pair (Ti, Tk) such that Ti and Tk have *conflicting* operations and the operation in Ti precedes that in Tk. The algorithm involves constructing a precedence graph for a given schedule S and looking for a cycle. If the graph is *acyclic,* then S is serializable; otherwise, it is nonserializable. The partial orders of the nodes in an acyclic precedence graph give the possible serial schedules that are equivalent to the given serializable schedule.

The schedules shown in Figs. 7–11 also show their respective precedence graphs. Consider the lost update problem in schedule S2 and its corresponding precedence graph. For the sake of illustration, we have labelled each edge with the data item followed by the sequence of conflicting operations. The conflicting pair of operations, T4:read(A) T3:write(A), causes the directed edge (T4, T3) to be drawn; similarly, the conflicting pair of operations, T3:write(A), T4:write(A), leads the directed edge (T3, T4). The cycle in the graph indicates that S2 is nonserializable. The precedence graph for serializable schedule S5, as expected, is acyclic (Fig. 11).

In our examples, there is an assumption that a write of a data item A is always preceded by a read of A; this is called ***constrained writes.*** In real applications, transactions may use ***unconstrained writes,*** that is, a write operation of a data item may appear independent of its read operation. The existence of unconstrained writes in transactions leads to the notion of ***view equivalence*** and ***view serializability,*** and a polygraph test for view serializability. The test for view serializability is $\mathcal{NP}$-complete. It can be shown that a conflict-serializable schedule is also view serializable, but not vice versa; that is, conflict serializability is more restrictive and may determine a view-serializable schedule (which contains unconstrained writes) to be nonserializable. We will not discuss view equivalence or view serializability here; interested readers are referred to Korth (13) and Papadimitriou (14).

Most database systems do not use these two concurrency control methods for imposing serializability, because it is practically impossible to determine beforehand how the operations of a schedule will be interleaved. Furthermore, when transactions are submitted continuously their boundaries are not clearly marked. If the serializability of a schedule is tested after transactions have committed, as the theory requires, then the effect of nonserializable schedules must be cancelled. This is a serious problem that makes this approach impractical. Therefore, the approach taken by most systems is to use a protocol that ensures serializability.

Concurrency Control Techniques

There are a number of concurrency control techniques that are used to ensure serializability or isolation of concurrently executing transactions. Some of the well-known techniques include locking, timestamp, and optimistic protocols; there are also multiversioned variations of the first two protocols.

Locking Protocols. The most widely used techniques for concurrency control are based on locking of data items. This enables access to data in a mutually exclusive manner; that is, when a transaction accesses a data item, then no other transaction can change it. Lock is a variable associated with a data item that describes the status of the data item with respect to read and write operations that can be applied to it. There are various modes in which a data item can be locked.

We discuss two of these locking modes, shared and exclusive. If data item A is locked in the ***shared*** mode by transaction T, then T can read A but cannot write A; a data item can be locked in the shared mode by *multiple* transactions, thus permitting *shared* read access to the data. If a data item is locked in the ***exclusive*** mode, then T can both read and write A; a data item locked in the exclusive mode cannot be locked in any mode by other transactions thus enforcing *exclusive* access to the data. A transaction unlocks the data item it has locked before it ends. A transaction that requests a lock on a data item that is locked in an incompatible mode must wait till it is able to acquire the lock. At any time a data item can be in any one of the three modes: unlocked, shared locked, or exclusive locked. Every transaction obtain an appropriate lock before reading or writing a data item.

One locking protocol that ensures serializability is the ***two-phase locking*** (2PL) protocol. This protocol requires that every transaction issue lock and unlock request in two phases. In the *growing phase,* a transaction can obtain locks but may not release any lock. In the *shrinking phase,* a transaction can release locks but may not obtain any new locks. Initially, a transaction is in a growing phase followed by a shrinking phase. It can be proven that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable, thus obviating the need for any test of serializability. It should be noted that it is possible that there are serializable schedules for a set of transactions that cannot be obtained through 2PL protocol.

There is a popular variation of the 2PL protocol described above called strict 2PL protocol. In this variation, a transaction does not release any lock until it commits or aborts. Any locking protocol can lead to a problem called deadlock. A ***deadlock*** occurs when two or more transactions are waiting for one another to release locks on some data items.

Timestamp Ordering Protocol. Timestamps are used to represent the order of transactions in a schedule. Transactions can be totally ordered according to their timestamps. The timestamp ordering rule is based on operations conflict discussed before. This scheme imposes a serial order on the participating transactions based on their timestamps and hence guarantees serializability. However, there are possible serializable schedules that are not allowed under the time stamp ordering protocol.

In this scheme, the system assigns a start timestamp to every transaction T denoted by TS(T). A transaction Ti is considered earlier than Tj, if $TS(Ti) < TS(Tj)$. Two variables are associated with each data item that represent the time of its last read and last write. This scheme uses the following read and write rules. A transaction's write request is valid only if that data item was last read and written by an earlier transaction. A transaction request to read a data item is valid only if the data item was written by an earlier transaction. If a transaction violates either of these two rules, then it must be aborted and later restarted.

Optimistic Concurrency Control. This protocol is called optimistic, since it is based on the observation that in some class of applications, the likelihood of two transactions accessing the same data item is low. Transactions are allowed to proceed as if there were no possibility of conflict with other transactions, and all data changes are applied to the local copies of each transaction. After the completion of the transaction, the protocol enters the validation phase and checks whether any of the transaction updates violated serializability. If serializability is not violated, then the database is updated using the transaction's private copy and the transaction is committed; otherwise, the transaction is aborted and restarted. The optimistic protocol described above maintains the start timestamp of a transaction, its read and write data sets, and the end timestamps of the various phases of the protocol.

Granularity of Data Item. As mentioned before, all concurrency control techniques assume that the database consists of a number of data items. The database item can be any one of the following: a field of a database record, a database record, a disk block or page, a file or table, or the entire database. Clearly, the larger the data item, the smaller the degree of concurrency but the lower the overhead of maintaining the locks. Most relational database systems provide the granularity of locks at the level of tuple (record) or disk block.

Levels of Isolation. The ISO and ANSI SQL standards mandate true isolation as the default, but few commercial systems follow this aspect of the standards, thus sacrificing correctness for performance. Relational database systems provide several levels of isolation, which can be chosen by the application for each transaction. These systems use short and long locks for implemeting various levels of isolation. A *short lock* on a data item is released right after the operation on that data item completes; a long lock, on the contrary, is released after the transaction has completed. We discuss next the ramifications of the four levels of isolation defined by the SQL2 standard.

Level 0. It is also called *browse, dirty read,* or *read uncommitted.* This is permitted for read-only transactions. As the name suggests, it allows a transaction to read other transactions' uncommitted data. No locks are set by the transaction running with this level.

Level 1. It is also known as *read committed* or *cursor stability.* Dirty reads and lost updates cannot occur in this mode. The system sets short shared locks on data that is read and long exclusive locks on data that is written.

Level 2. It is called *repeatable reads.* It does not have any of the three transactional dependencies. That is, it provides true isolation according to the theory of serializability discussed before. The system sets long shared locks on data that is read and long exclusive locks on data that is written.

Level 3. This is called *serializable.* This level subsumes level 2 and provides additional protection against phantom tuples. A stronger definition of repeatable reads demands that a transaction should not see an extra tuple—the *phantom tuple*—that is inserted in the middle of its two read operations and that satisfies its search criterion. This level may require a shared lock at the table or the predicate level (14).

Recovery

Recovery of failed or aborted transactions is an important capability provided by all sophisticated database systems. Recovery techniques are often closely tied to the concurrency control mechanism. Recovery from transaction failures means that the state of the database is restored to a correct state that existed in the past. In order to construct the correct

state, the system must keep information about changes made to the data items during transaction execution. This information is called the ***system log,*** which must be stored in the nonvolatile memory outside the database.

In case of noncatastrophic failures, the strategy for restoring the state of the database may either require undoing or redoing of some transactional operations. Both in the deferred and immediate update techniques, the updates are first persistently recorded in the system log before actually changing the database. The information recorded in the system log is crucial for recovery, and is used in both techniques.

SYSTEM ARCHITECTURE AND IMPLEMENTATION TECHNIQUES

The implementation of a DBMS varies from one system to another. To simplify the presentation, we focus on the implementation of a relational database management system (RDBMS).

DBMS System Architecture

The components within the dotted frame of Fig. 1 illustrate the internal component architecture of a DBMS. The ***stored data*** store database objects such as tables that are managed by the DBMS. The ***system catalog*** stores description of database objects. In the database, database objects are stored as ***files,*** which are logical abstractions of external storage devices. The abstraction allows the files to be accessed independently of the type of physical device. The query processor takes a database statement, which can be a DML or a DDL, generates an execution plan, and executes the plan (see Fig. 12). During the execution, whenever the query processor needs to transfer a database object to or from the database, it contacts the *data manager,* which transfers the higher level database object request into lower level file operations. The file operations are then performed by the *file manager.* The query processor, data manager, and file manager contact the system catalog for the description of database objects. The following subsections will describe each module in detail. The reader is referred to Refs. 6 and 8 for further reading on DBMS architecture.

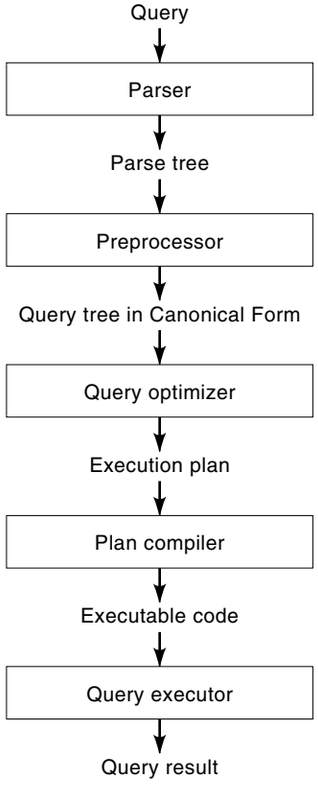


Figure 12. Query processing architecture.

Stored Data. Database objects stored in the stored data can include tables and indices. Indices are built for fast associative access to desired rows in a table. Their function is similar to that of an index for a book. Database objects are generally stored as files. For example, all rows in the table `PART` are stored as a file. An index built for the table `PART` is stored as another file.

System Catalog. The system catalog is also known as *metadata* or *data dictionary.* The database stores user data and the system catalog stores metadata, which includes a description of the database schema, the definition of tables and columns, and key and integrity constraints. For example, the system catalog stores the definition of the `PART` table, specifying the data types (e.g., part size is represented as an integer). It also stores the key constraint that `partkey` is the primary key of the table, and integrity constraints that it is involved in foreign key constraints with table `PARTSUPP` and `LINEITEM`.

Besides storing the description of database objects, the system catalog stores information needed by other modules in the DBMS, such as statistics of tables and columns, security and authorization specification, concurrency control information, and description of files in the database. For example, the statistics on the part size of the `PART` table, such as maximum and minimum part size, can be stored in the catalog and used by the query processor to generate an optimal execution plan. The security and authorization information are used by the data manager to control access permission for data objects to certain users. When concurrent accesses to the database are supported, the data manager uses the concurrent access information to ensure correct execution of these accesses. The file description is used by the file manager for accessing the database files.

Database Statements. *Database statements* include DML and DDL. Database statements are the only way users can modify or query the objects stored in the database. This was discussed in "Data Manipulation in the Relational Model."

Data Manager and File Manager

Storage Subsystem. The data manager and the file manager constitute the storage subsystem in a DBMS.

The ***file manager*** manages external storage devices such that objects stored on them can be accessed independently of the type of device (e.g., disks, RAM-disk, and tapes) and address.

An external storage device is partitioned into disjoint pages, and the read/write access to the device is performed in units of pages. For example, a page can be of size 2, 4, or 8-KB. Writing or reading one page to/from the device is called

one I/O (input and output). A database file consists of a number of pages.

The ***data manager*** takes a request to retrieve a row from the query processor, determines the page number on which the record resides, allocates a buffer in main memory to hold the page, and sends a request to the file manager to retrieve the page. A ***buffer*** is used for mapping a disk page into main memory. The file manager then determines the physical location of the page on the device, and retrieves the page into the allocated buffer. In case the desired page is already in a buffer, the data manager does not need to contact the file manager. It simply returns the row from the buffer to the query processor, and consequently no I/O is done. Access to the external storage device (also known as secondary storage device) takes much longer than main memory access. Thus buffering of data pages speeds up the DBMS considerably. A similar process occurs while writing a row to the device.

The query processor regards the database as a collection of records, the data manager regards the database as a collection of pages, and only the file manager knows cylinders, tracks, arms, and read/write heads of the device. Therefore, the file manager is the only component that is device dependent, and the remaining DBMS system is device independent. The mapping relationship is illustrated in Fig. 13. In some systems, the file manager is a component of the operating system, while others implement their own specialized file manager on raw disks.

The data manager contains the log manager, lock manager, transaction manager, and buffer manager. The ***buffer manager*** is responsible for making the buffer pages addressable in main memory, coordinating the writing of pages to disk with the log manager and transaction manager, and minimizing the number of actual disk I/Os. The system log is usually stored as a table, called log table. Each row in the log table records one read/write operation by a transaction. The ***log manager*** maintains and provides read and write access to the log table. The ***transaction manager,*** through interacting with the log manager and the lock manager, gathers information necessary in case of failure, and recovers error when failure occurs. The ***lock manager*** provides lock and unlock service for various lock requirements in the system. The data manager is responsible for ensuring the ACID properties for transactions. For more detail on implementation issues for these managers see to Ref. 2.

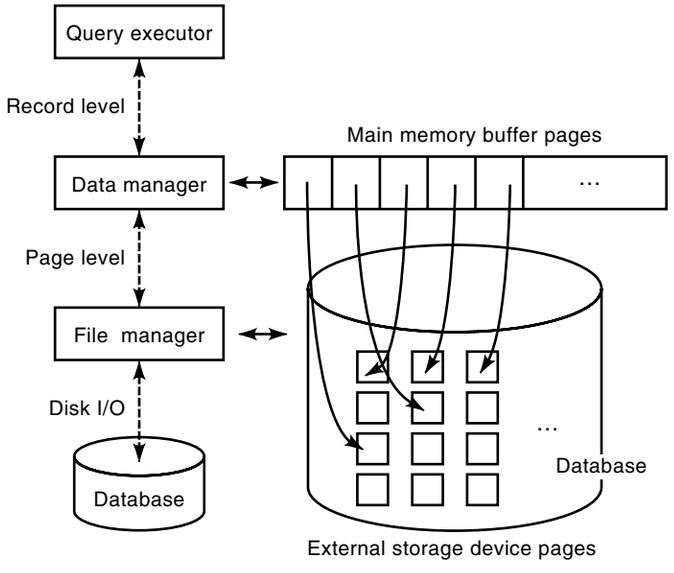


Figure 13. File and buffer manager.

Associative Access and Index. A DBMS allows associative access to a table to improve the efficiency of looking up rows in a table. Such access is accomplished by using indices on tables. An ***index*** consists of a set of index entries. Each ***index entry*** corresponds to a row in the table. An index entry consists of an index key and the rowid. A ***rowid*** stores the physical address of the row for this index entry on a secondary storage device. An index key consists of one or more column values in the table. The organization of the index entries depends on the type of index. The user can build multiple indexes on a table. An index is updated when its corresponding table is modified. There is a difference between a key of a relation as explained in the section on the "Relational Data Model. Basic Concept" and an index key. A key of a relation can uniquely identify a tuple, whereas the index key for a table is not necessarily unique. To avoid confusion, an index key that is unique is called a ***unique key*** and its corresponding index is called a ***unique index.*** There are two types of commonly used indices, B-tree and hash.

B-Tree Index. A ***B-tree*** index is a multilevel index. The first level of the index consists of index entries (key, pageid) where *pageid* is the address of a page and *key* is the first key of rows on the page. The second level of the index consists of index entries (key, pageid) where pageid is the address of a first-level index page and key is the first key value on the page, and so on until the highest level index entries can fit on a single page. The single page at the highest level is called the ***root*** and the lowest level pages are called ***leaf*** pages. To search for a row with key value k, we find a path from the root to some leaf where the desired row must reside if it exists. Then rows in the leaf are examined for a row with key k. The goal of the B-tree is quickly to find rows matching a particular range (or value) of the index key through a small number page reads. For example, for the query 'find parts that are of size 5 to 7,' the search starts at the root, where the key range is 5–11; then follow the index key range 5–7 in the next level, and consequently find the rowids for the first and third row in the table `PART`.

Hash Index. A transformation algorithm, *called* ***hash function,*** is used to transform the value of an index key into another value, called ***hash value.*** The transformation process is called *hashing.* An index key can consist of several columns but the hash value is a single value. The rowids of all rows in the table having the same hash value are stored on the same pages (called a ***hash bucket***) associated with the hash value. The hash index is useful for retrieving rows based on the value of its index key. For example, to build a hash index on column size for table `PART`, the hash function may hash 5 and 11 into the same hash value (bucket) 0, and hash 7 and 9 into the hash value (bucket) 1. Then, for the query 'find parts that are of size 5,' the DBMS hashes the key value 5 into hash bucket 0 using the same hash function, and finds the rowids of the two rows having size 5 and 11. It then compares the value 5 with the values of the size column of the

two rows and finds the rowid of the row with size 5. Based on the rowid, it retrieves and returns the row.

Clustered Index and Nonclustered Index. If you go to a library where books are placed on the shelves in order by authors' names, and want to find all the books written by Isaac Asimov, you would first look for Isaac Asimov in the index, then go to the shelves to find all his books. That is, the placement of the books (data rows) is determined by author name (index key value); this is called ***clustering.*** When the data rows of a table are stored in the order of the index key value, we call the index ***clustered index.***The advantage of a clustered index is that all data rows having the same index key values are likely to be stored on the same data pages, so after the first row is accessed, the data pages have been read into a buffer, and no more disk I/O is required for those other rows. Thus, the clustered index can significantly reduce read time for equality selection operation on the clustered index key. A B-tree clustered index can also reduce read time for range selection operation on the clustered index key.

Besides index maintenance, allowing efficient concurrent access to an index is also an important performance issue for OLTP applications. For discussion of this and other issues and other types of indices (e.g., bitmap indexes, grid file, k-d tree and R-tree indices) see Refs. 2, 15, and 16.

Query Processor

Figure 12 shows the general architecture of a ***query processor,*** which consists of the following steps: parsing, preprocessing, query optimization, plan compilation and query execution. Some systems may perform one step in several modules, while some may merge several steps. The ***parser*** checks the syntax of the input query and produces an internal representation called ***parse tree.*** The ***preprocessor*** takes the parse tree and produces an internal canonical representation called *query tree.* The ***optimizer*** takes the query tree, evaluates various query execution options, and produces an optimal query plan.

It can be shown that reordering the execution order of many relational operators under certain conditions will not change the result set. The optimizer mostly evaluates different execution orders of operators and different implementation algorithms for each operator. The *query plan* produced by the optimizer specifies the operator execution order along with the implementation algorithm for each operator. The ***plan compiler*** transforms the query plan into a form executable by the query executor. The ***executor*** then executes the plan and returns results. We describe these steps in detail below. We will use the following query as an example throughout this section:

Q5. Retrieve the name and type of the parts supplied by 'Jackson'.

```
SELECT PART.name, PART.type
FROM PART p, SUPPLER s, PARTSUPP ps
WHERE p.partkey = ps.partkey
      AND ps.suppkey = s.suppkey
      AND s.name = 'Jackson';
```

Commutativity and Associativity of Relational Operators. Many relational operators are commutative and associative. Here we described some of the important operator properties. We denote join as *, A, B, R as relations, and P, P1, P2, . . ., Pn as predicates.

Join is commutative and associative, so is the Cartesian product X

Cascade of selection (σ):

$$\sigma_{p1ANDp2AND\ldots ANDpn}(R) = \sigma_{p1}(\sigma_{p2}(\ldots(\sigma_{pn}(R)\ldots)$$

Commuting σ with join (and Cartesian product):

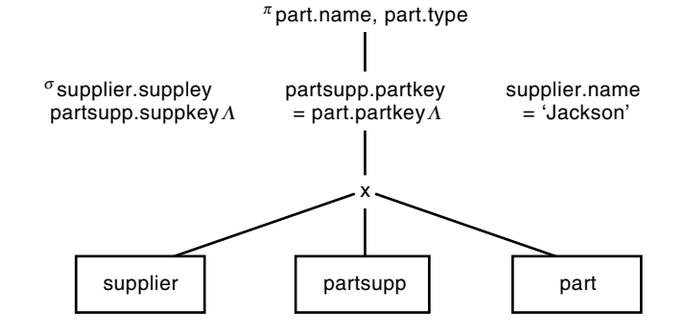
$$\sigma_p(A * B) = \sigma_p(A) * B \quad \sigma_p(A \times B) = \sigma_p(A) \times B$$

where P operates only on columns belonging to A.

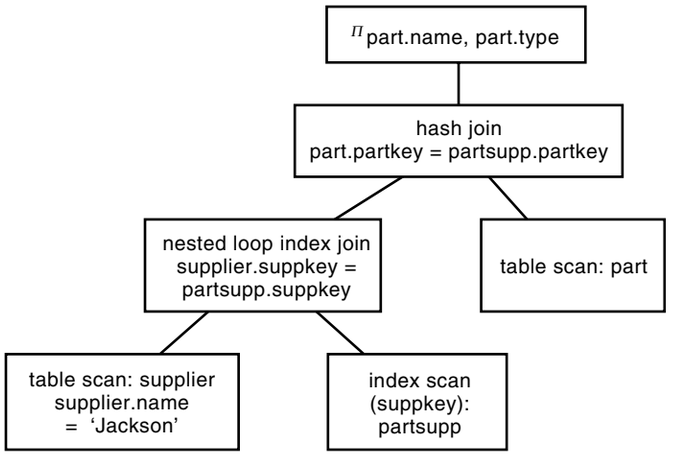
This means that we can freely permute the order of joins and single-table selections in a query. This is the foundation of the query-processing algorithms and optimization discussed in subsequent sections. For a given query with multiple joins and selections, the optimizer decides how to do each join and selection and the evaluation order of the joins and selections. For more description on the properties of relational operations, see Ref. 8.

Preprocessor. The ***preprocessor*** performs type checking, access permission validation, integrity constraint processing, view resolution and produces a canonical representation of the input query. *Type checking* enforces that the objects in the query, such as columns and aggregation functions, are referenced correctly. Access permission validation is done to ensure that the objects are accessible only to the authorized users. Type checking and access validation are done through enquiries to the system catalog. It then performs *view resolution* to expand views into the query. Then, the preprocessing step generates an internal representation, called ***query tree,*** of the query using a different form than that of the parse tree. Integrity constraints would then be added into the query tree.

Canonical Query Tree. There are generally many different ways of expressing a query in SQL. For example, there are several dozen ways of expressing Q5 in SQL. The performance of the query should not depend on how the user writes the query. Therefore, during preprocessing, query transformation algorithms are applied to the query tree to transform it into an equivalent ***canonical*** form. Two query trees are *equivalent* if they represent the same result set. The canonical form is neutral to any optimizer decision and thus allows any choices the optimizer makes. For example, Figure 14 shows a typical canonical query tree for query Q5. A ***query tree*** is a tree structure that represents tables as leaf nodes and relational algebra operators as internal nodes. The order of the operations is bottom-up: the lower level operation is performed first and feeds its result to its immediate higher level operation. In Fig. 14(a), the leaf nodes represent the operations of selecting all rows from the table PART, PARTSUPP, and SUPPLIER respectively. The second-level operation is a Cartesian product of the three result sets from selecting the three tables. The third-level operation is a selection operator, which takes the result set from the Cartesian product, and selects rows that satisfy the predicates. The final (topmost) operation is a projection operation that projects the result set from the selection operation immediately below it to produce the final query result. This query tree actually represents an inefficient way of executing query Q5. For example, if there



(**a**) Canonical query tree



(**b**) Query tree

Figure 14. Canonical query tree and query plan for query Q5.

are 100 rows in each table, the Cartesian product would generate 1 million rows before the selection operation is invoked. A better plan would be to perform some or all of the selection predicates as early as possible.

A good example of a canonical query tree can be found in Ref. 17 and more discussion on the canonical query tree can be found in Ref. 8.

Query Processing Algorithms. In a relational DBMS, algorithms for implementing all relational operators (e.g., selection, join, Cartesian product, group-by, order-by, and aggregation function) are required by the query executor to perform the operations. This section describes various such algorithms.

Access Paths. An *access path* implements the selection and projection operations. Several alternative access paths are available in a DBMS, and some may only apply with certain selection predicates. Table scan and index scan are two essential types of access paths.

A ***table scan*** retrieves all rows in a table, applies the selection predicates to the columns for each row, then returns qualifying rows. For example, for table `SUPPLIER` in query Q5, the executor retrieves all 5 rows in the table, applies the predicate, `name = ''Jackson''`, and returns the only qualifying row `<S12, Jackson, 11 Main St, S.F., 4155551212, 900.00>`. Therefore, a table scan needs to retrieve all pages of the table into main memory.

An ***index scan*** uses equality or range predicates to search the index entries and retrieves the data rows matching the predicates. Therefore, index scan retrieves only data pages that contain the qualifying rows. An index scan requires the presence of the search predicate to search the index entries. If the index is a hash index, then the index scan is a ***hash index scan.*** Similarly there is a ***B-tree index scan.*** For use of a hash index scan, there must exist an equality predicate, like "`name='Jackson'`", with the columns referenced being hash index key. Therefore, if such an index exists for `SUPPLIER`, then the above selection can be done by a hash index scan. Equality or range predicates on the index key are required for the B-tree index scan.

Join Methods. Join operation is one of the most time-consuming operations in query processing. Join is described in the section on "Relational Algebra." There are three commonly supported join methods to implement the join operator: nested-loop join, sort merge join, and hash join.

Nested loop join operates on two tables, the outer table and the inner table. For each row in the outer table, the algorithm retrieves all rows of the inner table, and outputs a result row for a match between an outer row and an inner row based on the join predicate. The optimizer decides the inner and outer order.

For example, for the join between table `SUPPLIER` and `PARTSUPP` in query Q5, assuming that the optimizer decides that `PARTSUPP` should be the outer table and `SUPPLIER` should be the inner table, the nested loop join proceeds as follows:

```
for each row in the PARTSUPP table
  for each row in the SUPPLIER table
    if (PARTSUPP.suppkey = SUPPLIER.suppkey),
     output the result row
```

The nested loop join scans the entire inner table once for each outer row. For example, if the outer table has one million rows, the inner table would be scanned a million times. Thus the nested loop join algorithm generally performs poorly compared to other join methods. The advantage of the nested loop join algorithm is that it applies to any type of joins, while other, more efficient join methods apply only to equi-joins.

The ***nested loop index join*** is a special case of nested loop join, in which the inner scan is always an index scan using an index key lookup on the join column. For example, for the same join as above, if the `SUPPLIER` table has an index on the column 'suppkey,' then the join can proceed as follows:

```
for each row p in the PARTSUPP table
  use p.suppkey to search index on SUPPLER.suppkey
    if match is found
      retrieve SUPPLIER row and output result
```

Therefore, an index nested loop join does not scan the entire inner table. It only scans the inner rows (and thus the inner data pages) and some index pages that match the outer row. The index used in the inner scan can be a temporary index that is created at query execution time and destroyed once the query is finished. The index nested loop join generally performs much better than the nested loop join, especially when the outer table is small and the inner table is very large.

The ***sort merge join*** only applies to equality join predicates and it requires that both inputs be sorted (ordered) on

the join columns before the join occurs. Both tables are scanned in order of the join columns. If there are no duplicate values in one of the joining columns, then the sort merge join reads both input interleavingly and return rows having the same value for the join columns. When there are duplicates on the join columns from both inputs, then the position in one input needs to back track when duplicates from the other input come in. There are some variations of the sort merge join. For example, both input tables may have an index on the joining columns, such that the scan on the inner table is an index key lookup.

The ***hash join*** also requires that the join predicate be an equality predicate. The most commonly used hash join algorithm works as follows. The optimizer decides that one input table should be a *probe table,* and the other input should be a *build table.* The build table is scanned and a hash index (hash table) is created on its join columns. Once the hash table is built, the probe table is canned and the join column of the probe rows is hashed using the same function. We then use the hash value to look up the hash table. Once a probe row hashes into a hash bucket containing some build rows, the join predicate is evaluated to find matches between the probe row and the build rows in the bucket. The matching rows are then returned. If the hash table cannot fit in main memory, it is partitioned into several hash tables (called partitions) such that each partition fits in memory. The probe input is then partitioned similarly and each probe partition is joined with its corresponding build partition. This is called hybrid hash join. For a complete description see Ref. 18. The major advantage of the hash join is that there is no requirement of order on join inputs. The hash join requires reading both tables entirely. Index nested loop join methods sometimes outperform the hash join because the index nested loop join does not need to access the entire inner table.

In most DBMSs, *joins of three or more tables* are performed by joining two tables first, and then joining the resulting intermediate table with the third table, and so on.

Aggregation Algorithms. Aggregation algorithms implement the group-by operator using hashing or sorting. Consider the following SQL query.

Q6. Find the total price for parts with the same size.

```
SELECT size, sum(price) FROM PART GROUP BY
size
```

Hash aggregation hashes the value of column 'size' for each input row and inserts the row into a hash table. Once the hashing process is finished, each hash bucket is examined and rows with the same value of 'size' are grouped together and the result is returned.

Sort aggregation sorts (orders) the input rows based on the value of 'size.' Once the sort is completed, the sorted stream is scanned and rows belonging to the same group are scanned consecutively and the result is returned.

Hash aggregation should generally outperform sort aggregation since sort aggregation is higher in computational complexity.

Readers are referred to Refs. 8 and 19 for more discussion on query processing algorithms.

Query Optimization. The process of generating an optimal execution plan is called ***query optimization.*** The optimizer decides the table access paths (e.g., whether to use an index) for each table in the query, join algorithms for each join, and algorithms for various other operations (e.g., order-by and group-by). In the case of a distributed database system, the optimizer determines the site where data resides and how to perform operations across sites. As mentioned earlier, many relational operations are commutative and associative; the optimizer also evaluates available operator evaluation orders and determines the optimal order. The optimizer produces the execution plan, which specifies the evaluation order of operators and an implementation algorithm for each operator.

Figure 14(b) shows an execution plan produced by the optimizer. The access paths for `PART, PARTSUPP`, and `SUPPLIER` are table scan, index scan using the index on `suppkey`, and table scan, respectively. While doing the scan on table `SUPPLIER`, the selection predicate "`SUPPLIER.name = 'Jackson'`" is applied to the supplier rows to eliminate unqualified rows; then a nested loop index join is used to join the intermediate result from the table scans of `SUPPLIER` (the outer table) with `PARTSUPP` (the inner table); then a hash join is used to join the intermediate result from the nested loop join (the build side) with the scan results for `PART` (the probe side); and finally, the resulting rows are projected on the desired columns to produce the final query result.

The optimizer generally uses a cost model to measure the cost of each query plan. The goal of the optimizer is to find the cheapest plan. With the permutation of choosing different algorithms for operators and different operator evaluation orders, there are an exponential number of possible query plans. A commonly adopted optimization objective is minimizing query resource consumption. The optimizer cost model is designed to measure the resource consumption. Lower cost means less resource consumption.

Query resources include CPU time and the number of disk I/Os required. In case of a distributed database, network communication cost is also considered one of the resources. Minimizing query resource consumption is generally adequate for minimizing the query response time for serially executed query plans, since serial execution does not allow intraoperator parallelism. Thus, even if a table is stored on two disks, the executor would scan one disk, and upon finishing, scan the second disk. Since CPU, network, and I/O costs are incomparable, each cost is given in predetermined weight ($w1$, $w2$, $w3$) so that the cost of an operation can be expressed as one single unit:

$$\begin{aligned}\text{operation cost} = {} & w1 * \text{CPU time} + w2 * \text{number I/O} \\ & + w3 * \text{network communication cost}\end{aligned}$$

The cost of each plan is the sum of the cost of each operation in the plan.

Optimizer Search Algorithm. The optimizer uses a ***search algorithm*** to search over all possible plans (called the ***search space***) and produces the plan it considers the cheapest. The most commonly used search algorithm is the dynamic programming algorithm. Essentially, it starts by building all smaller plan segments (called *partial plans*), then gradually building larger and larger partial plans until a complete plan is built and chosen. During the process of building these partial plans, the optimizer prunes more expensive partial plans. Property plays an important role during the process. A property is a description of the query result produced by a partial plan or a complete plan. It can be columns, sort order, predi-

cates that have been applied (and thus the result satisfies). No pruning is done if both competing partial plans have different properties. In summary, the algorithm first generates different access paths that contain different properties. Then, the algorithm generates all plans joining any two tables with different properties using all the access paths created in the first step. Then all plans joining three tables with different properties are built using all partial plans created in previous steps, and so on until a complete plan is generated. During the process, each partial plan with a distinct set of properties is built exactly once. For example, if there exists a B-tree index on column partkey for PART in query Q6, the optimizer would generate two access paths: one performs an index scan using the index and one uses a table scan. The index scan has the property that the scan result is sorted on column partkey, which can potentially be used in a sort merge join with PARTSUPP on the partkey. Therefore both partial plans are kept.

The reader is referred to Refs. 8, 19–22 for more discussion on query optimization.

Query Execution. The part of the query processor that performs the query execution is called ***query executor.*** There are generally two types of execution that the query executor needs to handle: ***data request*** and ***operational request.*** Data request includes the request to transfer database object to or from the database, such as a table scan or an index build. It may also require creation, deletion, or modification of a database object. The executor accomplishes data requests through the data manager. The operational request implements algorithms selected by the optimizer for the operators. It operates on the objects fetched from the database. Selection, projection, join, sorting, hashing, grouping, and aggregation are all operational requests. The operational requests are generally accomplished within the executor. The query executor operates in main memory and may use the external storage device-like disk as temporary storage area for operations requiring large amount of memory (e.g., sort and hash).

There are generally three types of query execution system, depending on the shape of plans it is capable of executing. The three types of plans are left deep, right deep, and bushy plans. The left deep plan allows only the outer table of a join to be an intermediate result, while right deep plans allow only the inner table of a join to be an intermediate result. While the left deep engine is common among existing commercial systems, it has been shown that right deep plans are more efficient when there is a large amount of main memory. Left deep or right deep engines simplify the query optimizer search space and execution engine, the bushy engine is more flexible, allowing either input of a join to be an intermediate result from another join.

The query executor uses an ***execution model*** to control data flow between operators, schedule operations, and provide a communication mechanism among operators. A good execution model should minimize communication cost between operators, simplify the communication mechanism and allow easy extension to the execution engine (like adding a new operator). The following describes a commonly used execution model called ***iterator mode.***

Iterator Model. The iterator model provides a generic approach for implementing various algorithms and scheduling mechanisms within the execution engine. An iterator has three functions: open, next, and close. It implements an operator, such as a table scan, index scan, nested loop join, or sort. The iterator for a binary operation has two input iterators, whereas the iterator for a unary operation has one input iterator. An iterator itself can be an input of another iterator, and may not have any input iterator (e.g., a scan iterator). A *parent iterator* has input iterators (called *child iterators*).

For example, for the table scan iterator, the open function is to open a table, the next function is to read the next row in the table, and the close function closes the table. When opening a table, the scan iterator opens the database file corresponding to the table, and prepares for retrieving a row. The next function of the scan iterator then reads a row and returns to its parent. The parent iterator calls the next function repeatedly until there is no more data row available. Then it calls the close function to close the database file and finish the scan. Therefore the scan iterator has no input iterator. In Fig. 14, each box represents an iterator implementing each operator.

Some parent iterators require only one row from their child iterators before they start their own execution. They are called ***nonblocking iterators.*** A nonblocking iterator processes a row as soon as it is returned from its input iterators. And if its parent is also a nonblocking iterator, its parent also processes the row immediately. The process can possibly cascade up to the root of the execution tree. One kind of parent iterator (the ***blocking iterator***) requires all rows from its child before it can start processing. Table scan, index scan, nested loop join, and nested loop index join are all nonblocking iterators. Sort merge join, hash join, and aggregation are blocking iterators. Therefore, nonblocking iterators do not require temporary memory for storing the rows they receive, while blocking iterators do.

For parallel execution of an iterator tree, an iterator called an exchange iterator is inserted between two iterators. The exchange iterator does not perform any data manipulation. Its sole responsibility is to provide data redistribution, process management, and flow control between the two iterators.

An iterator schedules itself, and the entire query plan is executed within a single process. Communication between iterators is done through function calls and is inexpensive. Each iterator produces one intermediate result row at a time on demand from its parent. The iterator can schedule any type of trees, including bushy trees. Each iterator is a self-sufficient entity that does not need to understand the complexity of other iterators in the plan. Adding one new iterator does not require changes to the existing iterators. Thus an execution engine using the iterator model can be extended by simply adding new iterators. For a more detailed description of the iterator model the reader is referred to Ref. 23. Graefe (18) gives a survey of query processing techniques.

ADVANCED TOPICS

Data Mining

Data mining is used to extract patterns or rules from large databases automatically. It combines techniques from machine learning, pattern recognition, statistics, databases, and visualization. Data mining has become important because of several factors. The cost of computing and storage is now low enough that companies can collect and accumulate detailed data about their business. Further, data warehousing tech-

niques have enabled the consolidation of all data needed for analysis into a single database. Lastly, intense competition is leading businesses to look for new ways of gaining insight into their businesses in the hope of discovering some competitive advantage.

Data mining may be used to derive several kinds of abstractions. Some examples are association rules, classification, and clustering. An example of an association rule is "if a customer buys milk and bread she/he also buys eggs." The importance of such a rule is measured by how often milk and bread are bought together (called the support for the rule) and the fraction of purchases of milk and bread in which eggs are also purchased (called the confidence). Data mining techniques are available that, given the minimum acceptable support and confidence, can be used to find all association rules of the form "if a customer buys X and Y he/she also buys Z."

Classification is the division of data into classes based on the values of some attributes. The system is first trained by the use of a set of training objects. After training is complete, new objects may be classified. For example, a credit approval application may be trained using credit data for cases whose outcome is known.

Clustering requires discovering the criteria for dividing data into new classes. Data are clustered into classes based on their features with the objective of maximizing intraclass similarity and minimizing interclass similarity.

Data Warehouse and OLAP

A ***data warehouse*** is a subject-oriented, integrated, time-varying, and consistent collection of data used primarily in organizational decision making. It is a popular approach for implementing a decision support system (DSS). *Data warehousing* is a collection of decision support techniques. It is mainly used in an organization by executives, managers, and analysts to make faster and better decisions. Data warehouses are implemented on a DBMS called the *data warehouse server*. Figure 15 shows the architecture of a data warehouse. The data warehouse server extracts data from various sources, which can be an OLTP DBMS within the organization, or other sources such as flat files and spread sheets. It then cleans the data and transforms them into a desirable format. Data from different sources are then integrated into a single database schema and stored in the data warehouse. Therefore, the data stored in data warehouse are historical and derived data. The data warehouse server also maintains a metadata repository. The data are periodically refreshed at certain time intervals. Users access the data through a variety of front-end applications, such as query and report, planning and analysis, and data mining. These applications often have graphical user interfaces. Query and reporting application allows users to query the data warehouse and generate reports. Planning and analysis address essential business problems such as budgeting, forecasting, sale analysis, what-if analysis, and financial analysis. Data mining application allows users to obtain patterns or rules for the data automatically.

A data warehouse is generally modeled using a star schema or a snowflake schema, as shown in Fig. 16. A ***star schema*** (shown in Fig. 16 inside the dotted line) consists of a fact table and a number of dimensional tables. The fact table is very large and contains detailed information for each record in the data warehouse. A dimensional table describes an attribute in the fact table. For example, the `PART` table describes each part in the `LINEITEM` table. A dimensional table and the fact table maintain a foreign key relationship. When a certain dimension needs lower level information, the dimension is modeled by a hierarchy of tables, such as the data, month, and year tables in Fig. 16. The schema is then called snowflake schema. Thus, data in the data warehouse are modeled multidimensionally, with each dimension corresponding to a dimension table.

On-line analytic processing (OLAP) is a data warehousing technique based on multidimensional modeling of the organizational data. Rather than viewing the data as tables of records, OLAP introduces a multidimensional data model that is believed to be more intuitive for nontechnical knowledge workers. Data are viewed as a multidimensional cube. Each dimension of the cube represents an attribute and each cell contains a numeric or summary attribute. For example, we may have sales volume as a summary attribute with di-

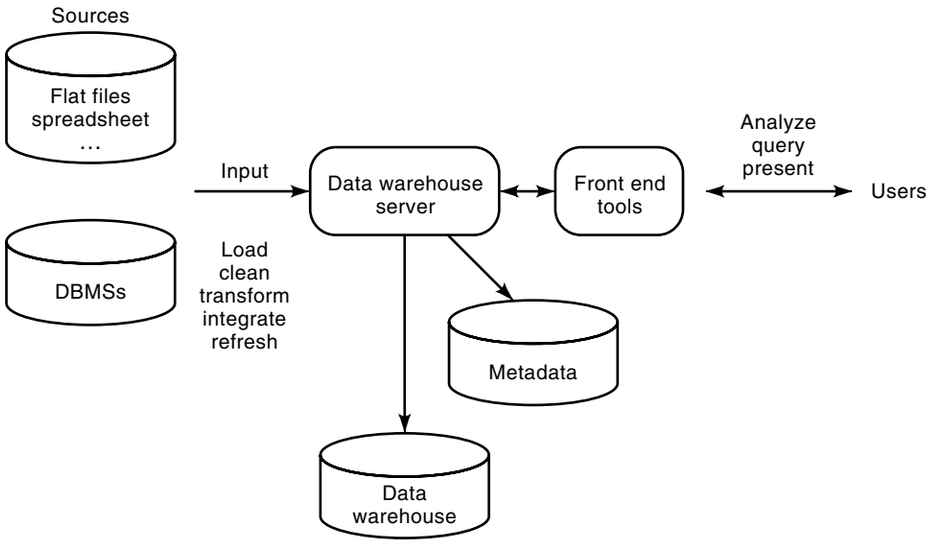


Figure 15. Data warehouse architecture.

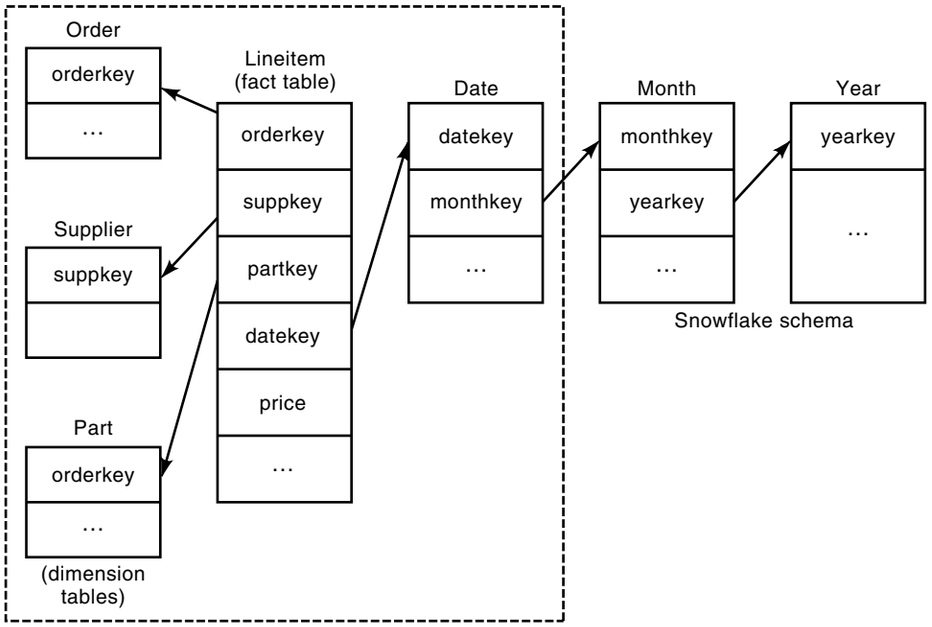


Figure 16. Star and snowflake schemas.

mensions such as product, time, and geography. A dimension can be hierarchical. Time may be viewed in units of years, quarters, months, weeks, or days. Geography may be viewed as country, region, city, or even individual sales offices. A query may be posed, for example, to find the total sales volume for each product for each country in the month of December 1997.

The OLAP model defines certain operations on data cubes. Roll-up is used to summarize data. Drill down is used to go from higher level summary data to lower level data. For example, for a particular product, we may find the detailed sales data for each office on a certain data. Slicing and dicing are similar to the selection and projection operators of the relational model. Pivoting is used to reorient a cube.

The data warehouse server can be a standard relational DBMS, an OLAP server using a relational DBMS for storing its data (ROLAP), or a multidimensional OLAP (MOLAP) server that stores multidimensional data in special format using its own data storage subsystem. The ROLAP server can take advantage of the strength of relational DBMS which is capable of handling large size data warehouse containing terabytes of data.

One of the key issues in data warehousing is how to index the data so that complex queries can be answered quickly. Bitmap indexes are used to speed up selection, projection, and aggregation. Bitmap indexes use bit vectors to represent rows and column values. This provides fast access for a class of queries but is inefficient for data modification operations. (Reference 15 has a detailed description of bitmap indexes.) A join index maintains the relationships among two or more tables and is essentially a precomputed join. Thus it can speed up join operations, but the index is expensive to maintain since data modification to one of its member tables could result in change of the index.

Materialized view is another important research topic for data warehousing. The issues are deciding the views to materialized, exploiting the views for answering queries, and view maintenance. (See Refs. 24–26 for a detailed description.) Complex query optimization is another important issue. An OLAP server needs to handle queries that contain aggregation and subqueries. Subquery flattening and commuting aggregation and joins are two important techniques. (References 27–31 have detailed descriptions of these techniques.) Other research issues include approaches for fast loading of data, data cleaning techniques, data warehouse management tools, data warehouse design tools, extension of SQL to support special requirement of OLAP queries. For other issues and more detailed description on data warehousing see Refs. 32 and 33.

Active Databases

Active database systems support rules (called production rules) that specify data manipulation operations (called actions) to be executed automatically whenever certain events occur or conditions are satisfied. The rules provide a powerful mechanism for supporting such DBMS features as integrity constraint envorcement, view maintenance, and access authorization. Active database systems are DBMS that support production rules. Active database systems also provide a powerful platform for implementing large and sophisticated expert systems and knowledge base systems. For example, users can specify a rule like "delete a supplier from the `SUPPLIER` table when the supplier does not supply parts anymore." Some systems call rules as triggers. A rule can take the form: **`on`** `event` **`if`** `condition` **`then`** `action`.

The rule is triggered when the event occurs. Once the rule is triggered, the condition is checked on the data and, if satisfied, the action is performed. Examples of events include data modification (like insert, update, delete rows or tables), data retrieval (select), and timing. For example, when a row (part) is deleted from the `PART` table, a rule may be specified to delete all rows in the `PARTSUPP` table that record the suppliers

for the part. Rules with timing event may be triggered at certain time intervals or times. Conditions include query, predicates over the database states, and predicates over the change in the database states. When the condition is a query, it usually returns true when the query returns any data. The condition part can be empty so that the event always triggers the action. The actions of the rules can be data modification, data retrieval, rollback or abort of the current transaction, or even sending an email. Active DBMS allow the user to specify how to resolve conflict, that is, the choice of the rule to be executed when multiple rules are triggered.

Important research issues include improving the expressive power of rules, efficient maintenance of rules, rule processing, conflict resolution methods, the semantics of error recovery during rule processing, deadlock avoidance or resolution during rule execution, methods for ensuring DBMS performance with the present of rules, smooth integration of the rule system with the DBMS, real time monitoring, support for application development, and parallel execution of rules. For a more detailed description, see Refs. 34 and 35.

Extended Transaction Models

The current state of the art in transaction processing is characterized by the classical transaction model discussed in the section on "Transaction and Concurrency Control." These transactions focus on ACID properties, are *flat,* and provide a *single* execution framework. This model has the great advantage of conceptual and formal simplicity, and it has proved to be a powerful and widely accepted concept. However, applications are getting more complex, integrated and sophisticated, and their needs are far from being well served by the classical transactions. As a result, many extended models have been proposed. They permit the modeling of higher level operations and exploit application semantics. In addition to the extension of internal transactional structure, they seek to provide selective relaxation of atomicity and isolation properties.

A common extension of flat transactions is ***nested transactions*** (36), which is a set of subtransactions that may recursively contain other subtransactions, thus forming a transaction hierarchy. Nested transactions may provide full isolation at the global level, but they permit increased modularity and finer granularity of failure handling; complex interactions (37) may take place between a transaction and its subtransactions, while the top-level transaction retains final control of overall commitment and rollback. *Savepoints* are a special case of nested transactions. Several commercial relational database systems provide savepoints, or a simple form of nested transactions. ***Chained transactions*** allow for committing certain stable, intermediate results so that they will not be lost in case of system failure, while still keeping control over resources that should not be allocated to other transactions; the chained transactions can be categorized as flat transactions. ***Multilevel*** *transactions* are a variant of nested transactions that allow for an early commit of intermediate results of lower levels inside the transaction while isolation is still controlled at higher levels, provided that there are counter (or compensating) actions to the committed result that can be executed in case of a rollback. ***Long-lived transaction*** is an important class of transactions that generally have three characteristics: minimization of lost work due to system or program failures, recoverable computation, and explicit control flow (2). *Minimization of lost* work is achieved by durably storing of parts of a transaction without compromising isolation. *Recoverable computation* is needed for transactions that take days or weeks but still represent one unit of work that must be organized so that the transaction can be suspended and resumed. Note that in the classical model of transaction there is no notion of suspending a transaction that can survive system shutdown/restart. *Explicit control flow* requires that transactions be able either to proceed by correcting the changes or to discard all transactional changes, including the durable ones.

Sagas (38) were introduced to deal with long-lived transactions. Sagas are linear sequences of transactions with a predefined order of execution and a corresponding set of compensating transactions; a saga completes successfully if all the subtransactions are committed; if any one of the subtransactions fails, then all its preceding committed subtransactions are undone by executing their corresponding compensating subtransactions. A method for implementing long-running transactions (called *work-in-progress activities*) on top of a relational database system is described in Ref. 39; Subtransactions can be durably committed in this scheme, but their effect remains invisible to the outside observer; in case of failure, work-in-progress activity allows undoing of committed subtransactions without requiring a separate component for compensating transactions; it provides minimization of lost work, recoverable computation, and explicit control flow. Work-in-progress activity as well as sagas provide increased transaction concurrency by relaxing the requirement for strict isolation. Other proposals include *migrating transactions* (40) and *flexible transactions* (41). Strict isolation is easy to implement, but quite restrictive in some cases and unacceptable for long-running activities. A related area of active research involves extending the classical transactional model by describing dependencies that arise on shared data during concurrent execution. There are proposals (42,43), for this model of transaction that preserve invariants over database.

Spatial Database

Spatial data is a term used to describe spatial objects made up of points, lines, regions, surfaces, and polygons. Spatial data can be discrete or continuous. Examples of spatial data include maps of cities, rivers, roads, mountain ranges, and parts in a computer-aided design (CAD) system. New application areas that require storing and querying of spatial data include geographic information systems (GIS), CAD, computer-aided manufacturing (CAM), remote sensing, environmental modeling, and image processing.

There are several levels at which queries to spatial data can be described. At the highest level, the most common queries are to display the data, to find a pattern in the data, or predict the behavior of data at another location. Another class of query is polygon or simple overlay, which requires an operation that may be termed a spatial join. Focal queries include search, proximity determination, and interpolation.

One of the key issues in building a spatial database management system (44) is deciding how to integrate spatial and nonspatial data. Many researchers use the classifications *dedicated, dual,* and *integrated* for different architectures. Dedicated systems are built to support only spatial data and therefore are not extensible. Dual architectures are based on

distinguishing between spatial and nonspatial data by using different data models. Dual architecture implies the existence of two storage managers; there are problems such as locking integrity and synchronization in this scheme. An integrated architecture is more general. It involves extending nonspatial database systems with their own abstract data type and efficient access methods for these data types. Query optimization in spatial databases is a relatively underdeveloped field. A framework for different optimization strategies and spatial access methods is needed. There are issues such as the effect of different representations, spatial access methods, and clustering and connectivity of the data sets. Interested readers are referred to Ref. 44.

Temporal Databases

Conventional database management systems capture the current snapshot of reality. Although they serve some applications well, they are inadequate for those applications that directly or indirectly refer to time. A temporal database fully supports the storage and querying of information that varies over time. Considerable research effort has been directed to temporal databases and temporal aspects of information management. A taxonomy (45) of time in databases has been developed.

In fact, most applications require temporal data to a certain extent. Conventional database systems may be able to meet this need at the expense of higher data redundancy, inelegant modeling, and cumbersome query languages. Therefore, a cohesive and unified formalism is needed to manage and manipulate temporal data. Many temporal extensions to relational and object-oriented database systems and their associated query languages have been proposed. Extensions to the relational model fall into two broad categories, tuple timestamping and attribute timestamping. In the former case, a relation is augmented with two timestamps, representing a time interval during which the values in the tuple are relevant. In the latter case, timestamps are added to attributes, thus changing the domain of possible values. Tuple timestamping, as it remains within the framework of first normal form relations, benefits from all the advantages of traditional relational database technology. Attribute timestamping, on the other hand, requires non–first normal form (nested) relations, which are more difficult to implement. Although attribute timestamping may add some modeling capability, the representation of temporal relationships remains a problem in this model.

The increase in the complexity of new applications such as computer-aided design, scientific and multimedia databases has led to the temporal extensions to the object-oriented and deductive data models. The issue of capturing many implicit representations of temporal data instances in temporal object-oriented databases still needs investigation. For example, tuple timestamping using time intervals (temporal elements) assumes that the values of the temporal attributes remain constant within an interval. However, if this is not true, then one must resort to explicit representation of each time-point value, which may be either impossible or too expensive.

The other area of interest is optimization of temporal queries and specialized access methods. Temporal queries are more involved than conventional queries. The relations over which temporal queries are defined are larger and grow monotonically; furthermore, the predicates in temporal queries are complex and harder to optimize. Temporal overlap and inequality comparisons are quite common. Interested readers are referred to Tansel (46) for a comprehensive discussion on this topic.

BIBLIOGRAPHY

1. M. Stonebraker, *Object-Relational DBMSs,* San Francisco, CA: Morgan Kaufmann, 1996.
2. J. Gray and A. Reuter, *Transaction Processing,* San Francisco, CA: Morgan Kaufmann, 1993.
3. J. Gray (ed.), *The Benchmark Handbook for Database and Transaction Processing Systems,* San Mateo, CA: Morgan Kaufmann, 1991.
4. Transaction Processing Council homepage http://www.tpc.org.
5. E. F. Codd, A relational model for large shared data banks, *Comm. ACM,* **13** (6): 377–387, 1970.
6. J. Ullman, *Database and Knowledge-base Systems,* Rockville, MD: Computer Science Press, 1988.
7. D. Maier, *The Theory of Relational Databases,* Rockville, MD: Computer Science Press, 1983.
8. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems,* Redwood City, CA: Benjamin/Cummings, 1994.
9. C. J. Date and H. Darwen, *A Guide to the SQL Standard,* Reading, MA: Addison-Wesley, 1993.
10. R. G. G. Cattell, *Object Data Management,* Reading, MA: Addison-Wesley, 1991.
11. E. F. Codd, Extending the database relational model to capture more meaning, *ACM Trans. Database Systems,* **4** (14): 397–434, 1979.
12. W. Kim (ed.), *Modern Database Systems,* Reading, MA: Addison-Wesley, 1995.
13. H. F. Korth and A. Silberschatz, *Database System Concepts,* New York: McGraw-Hill, 1986.
14. C. Papadimitriou, *The Theory of Database Concurrency Control,* Rockville, MD: Computer Science Press, 1986.
15. P. O'Neil and Dallan Quass, Improved query performance with variant indexes, *SIGMOD Record,* **26** (2): 38–49, 1997.
16. A. Guttman, R-Trees, A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. Manage.,* 1984, pp. 47–57.
17. H. Pirahash, J. M. Hellerstein, and W. Hasan, Extensible/rule base query rewrite optimization in Starburst, *Proc. ACM SIGMOD Conf.* 1992, pp. 39–48.
18. G. Graefe, Query evaluation techniques for large databases, *ACM Comput. Surveys,* **25** (2), 70–170, 1993.
19. P. O'Neil, *Database Principles Programming Performance,* San Francisco, CA: Morgan Kaufmann, 1994.
20. P. Selinger, Access paths selection in a relational database management system, *Proc. ACM SIGMOD Conf.,* 1979, pp. 23–34.
21. G. M. Lohman, Grammar-like functional rules for representing query optimization alternatives, *SIGMOD Record,* **17** (3): 18–27, Conf. 1988.
22. K. Ono and G. Lohman, Measuring the complexity of join enumeration in query optimization, *Proc. VLDB Conf.,* pp. 314–325, 1990.
23. G. Graefe, Encapsulation of parallelism in the volcano query processing system, *Proc. ACM SIGMOD Conf.,* 1990, pp. 102–111.
24. H. Z. Yang and P.-A. Larson, Query transformation for PSJ-queries, *Proc. 13th VLDB Conf.,* 1987, pp. 245–254.

25. I. S. Mumick, D. Quass, and B. S. Mumick, Maintenance of data cubes and summary tables in a warehouse, *Proc. ACM SIGMOD Conf.,* 1997, pp. 100–111.
26. V. Harinarayan, A. Rajaraman, and J. Ullman, Implementing data cubes efficiently, *SIGMOD Record,* **25** (2): 205–216, 1996.
27. W. Kim, On optimizing a SQL-like nested query, *ACM Trans. Database Syst.,* **7** (13): 443–469, 1982.
28. R. Ganski and H. K. T. Wong, Optimization of nested SQL queries revisited, *SIGMOD Record,* **16** (3): 23–33, 1987.
29. U. Dayal, Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregations and quantifiers, *Proc. VLDB Conf.,* 1987, pp. 197–208.
30. Muralikrishna, Improved unnesting algorithms for join aggregation SQL queries, *Proc. VLDB Conf.,* 1992, pp. 91–102.
31. W. Yan and P.-A. Larson, Eager aggregation and lazy aggregation. *Proc. VLDB Conf.* 1995, pp. 345–357.
32. W. H. Inmon, Building the data warehouse, New York: Wiley, 1992.
33. S. Chaudhuri and U. Dayal, Overview of data warehousing and OLAP technology, *SIGMOD Record,* **26** (1): 65–74, 1997.
34. J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing,* San Francisco, CA: Morgan Kaufmann, 1996.
35. E. N. Hanson and J. Widom, An overview of production rules in database systems, *Knowledge Eng. Rev.,* **8** (2): 121–143, 1993.
36. J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing,* Cambridge, MA: MIT Press, 1985.
37. U. Dayal, M. Hsu, and R. Ladin, A transaction model for long running activities, *Proc. 17th VLDB Conf.,* 1991, pp. 113–122.
38. H. Garcia-Molina and K. Salem, Sagas, *SIGMOD Record,* **16** (3): 249–259, 1987.
39. R. Ahmed and U. Dayal, Management of Work-in-progress in Relational Systems, *Proc. of 3rd IFCIS Conf. Cooperative Inf. Syst.,* New York, 1998.
40. J. Klien and A. Reuter, Migrating transactions, *Workshop Future Trends Distributed Comput. Syst.,* 1988, pp. 512–520.
41. Y. Leu, A. Elmargarmid, and M. Rusinkiewics, *An Extended Transaction Model For Multidatabase Systems,* Purdue University, CSD-TR-925, 1989.
42. P. Peinl, A. Reuter, and H. Sammer, High contention in a stock trading database: A case study, *SIGMOD Record,* **17** (3): 260–268, 1988.
43. P. E. O'Neil, *Escrow Promises,* Boston: Univ. Massachussetts, 1990.
44. H. Samet, *The Design and Analysis of Spatial Data Structures,* Reading, MA: Addison-Wesley, 1990.
45. K. K. Taha, R. T. Snodgrass, and M. D. Soo, Bibliography on spatio-temporal databases, *SIGMOD Record,* **22**: (1), 59–67, 1993.
46. A. U. Tansel, (ed.), *Temporal Databases,* Redwood City, CA: Benjamin/Cummings, 1993.

RAFI AHMED
WAQAR HASAN
WEIPENG YAN
Informix Software, Inc.

DATABASE SCHEMAS. See DATABASE MODELS.
DATABASES, DEDUCTIVE. See DEDUCTIVE DATABASES.
DATABASES, DISTRIBUTED. See DISTRIBUTED DATABASES.
DATABASES, MULTIMEDIA. See MULTIMEDIA INFORMATION SYSTEMS.
DATABASES, RELATIONAL. See RELATIONAL DATABASES.
DATABASES, SPATIAL. See SPATIAL DATABASES.
DATABASES, TEMPORAL. See TEMPORAL DATABASES.
DATABASE TRANSACTION. See TRANSACTION PROCESSING.
DATA CLASSIFICATION. See DATA ANALYSIS; DATA REDUCTION.