Oracle Data Warehouse Management

Secrets of Oracle Data Warehousing

Mike Ault

Retail Price $12.95 US/$19.95 Canada
ISBN: 0-9740716-4-1

Rampant TechPress

Oracle Data Warehouse Management

Secrets of Oracle Data Warehousing

#
#
#

Mike Ault

Notice

While the author & Rampant TechPress makes every effort to ensure the information presented in this white paper is accurate and without error, Rampant TechPress, its authors and its affiliates takes no responsibility for the use of the information, tips, techniques or technologies contained in this white paper. The user of this white paper is solely responsible for the consequences of the utilization of the information, tips, techniques or technologies reported herein.

Oracle Data Warehouse Management

Secrets of Oracle Data Warehousing

By Mike Ault

Published by Rampant TechPress, Kittrell, North Carolina, USA

Series Editor: Don Burleson

Production Editor: Teri Wade

Cover Design: Bryan Hoff

Visit **www.rampant.cc** for information on other *Oracle In-Focus* books.

ISBN: 0-9740716-4-1

Table Of Contents

Introduction

I am Michael R. Ault, a Senior Technical Management Consultant with TUSC, an Oracle training, consulting and remote monitoring firm. I have been using Oracle since 1990 and had several years of IT experience prior to that going back to 1979. During the 20 odd years I have been knocking around in the computer field I have seen numerous things come and go. Some were good such as the PC and all it has brought to the numerous languages which have come, flared briefly and then gone out.

Data warehousing is a concept that really isn't new. The techniques we will discuss today have their roots back in the colossal mainframe systems that were the start of the computer revolution in business. The mainframes represented a vast pool of data, with historical data provided in massive tape libraries that could be tape searched if one had the time and resources.

Recent innovations in CPU and storage technologies have made doing tape searches a thing (thankfully) of the past. Now we have storage that can be as large as we need, from megabytes to terabytes and soon, petabytes. Not to mention processing speed. It wasn't long ago when a 22 mghz system was considered state-of-the-art, now unless you are talking multi-CPU each at over 400 mghz you might as well not even enter into the conversation. The systems we used to think where massive with a megabyte of RAM now have gigabytes of memory. This combination of large amounts of RAM, high processor speed and vast storage arrays has led to the modern data warehouse where we can concentrate on designing a properly architected data structure and not worry what device we are going to store it on.

This set of lessons on data warehousing architecture and Oracle is designed to get you up to speed on data warehousing topics and how they relate to Oracle. Initially we will cover generalized data warehousing topics and then Oracle features prior to Oracle8i. A majority of time will be spent on Oracle8 and Oracle8i features as they apply to data warehousing.

Hour 1:

Conceptual Overview

Objectives:

The objectives of this section on data warehouse concepts are to:

1. Provide the student with a grounding in data systems architectures
2. Discuss generic tuning issues associated with the various data systems architectures.

Data Systems Architectures

Using the proper architecture can make or break a data warehouse project.

OLTP Description and Use

OLTP Stands for On-Line Transaction Processing. In an OLTP system the transaction size is generally small affecting single or few rows at a time. OLTP systems generally have large numbers of users that are generally not skilled in query usage and access the system through an application interface. Generally OLTP systems are designed as normalized where every column in a tuple is related to the unique identifier and only the unique identifier.

OLTP systems use the primary-secondary key relationship to relate entities (tables) to each other.

OLTP systems are usually created for a specific use such as order processing, ticket tracking, or personnel file systems. Sometimes multiple related functions a re performed in a single unified OLTP structure such as with Oracle Financials.

OLTP Tuning

OLTP tuning is usually based around a few key transactions. Small range queries or single item queries are the norm and tuning is to speed retrieval of single rows. The major tuning methods consist of indexing at the database level

and using pre-tuned queries at the application level. Disk sorts are minimized and shared code is maximized. In many cases closely related tables may be merged (denormalized) for performance reasons.

A fully normalized database usually doesn't perform as well as a slightly de-normalized system. Usually if tables are constantly accessed together they are denormalized into a single table. While denormalization may require careful application construction to avoid insert/update/delete anomalies, usually the performance gain is worth the effort.

OLAP Description and Use

An OLAP database, which is an On-line Analytical Processing database, is used to perform data analysis. An OLAP database is based on *dimensions* a dimension is a single detail record about a data item. For example, a product can have a quantity, a price, a time of sale and a place sold. These four items are the dimensions of the item product in this example. Where the dimensions of an object intersect is a single data item, for example, the sales of all apples in Atlanta Georgia for the month of May, 1999 at a price greater than 59 cents a pound. One problem with OLAP databases is that the cubes formed by the relations between items and their dimensions can be sparse, that is, not all intersections contain data. This can lead to performance problems. There are two versions of OLAP at last count, MOLAP and ROLAP. MOLAP stands for Multidimensional OLAP and ROLAP stands for Relational OLAP.

The problem with MOLAP is that there is a physical limit on the size of data cube which can be easily specified. ROLAP allows the structure to be extended almost to infinity (petabytes in Oracle8i). In addition to the space issues a MOLAP uses mathematical processes to load the data cube, which can be quite time intensive. The time to load a MOLAP varies with the amount of data and number of dimensions. In the situation where a data set can be broken into small pieces a MOLAP database can perform quite well, but the larger and more complex the data set, the poorer the performance. MOLAPs are generally restricted to just a few types of aggregation.

In a ROLAP the same performance limits that apply to a large OLTP come into play. ROLAP is a good choice for large data sets with complex relations. Data loads in a ROLAP can be done in parallel so they can be done quickly in comparison to a MOLAP which performs the same function.

Some applications, such as Oracle Express use a combination of ROLAP and MOLAP.

The primary purpose of OLAP architecture is to allow analysis of data whether comes from OLTP, DSS or Data warehouse sources.

OLAP Tuning

OLAP tuning involves pre-building the most used aggregations and then tuning for large sorts (combination of disk and memory sorts) as well as spreading data across as many physical drives as possible so you get as many disk heads searching data as is possible. Oracle parallel query technology is key to obtaining the best performance from an OLAP database. Most OLAP queries will be ad-hoc in nature, this makes tuning problematic in that shared code use is minimized and indexing may be difficult to optimize.

DSS Description and Use

In a DSS system (Decision Support System) the process of normalization is abandoned. The reason normalization is abandoned in a DSS system is that data is loaded and not updated. The major problem with non-normalized data is maintaining data consistency throughout the data model. An example would be a person's name that is stored in 4 places, you have to update all storage locations or the database soon becomes unusable. DSS systems are LOUM systems (Load Once – Use Many) any refresh of data is usually global in nature or is done incrementally a full record set at a time.

The benefits of an DSS database is that a single retrieval operation brings back all data about an item. This allows rapid retrieval and reporting of records, as long as the design is identical to what the user wants to see. Usually DSS systems are used for specific reporting or analysis needs such as sales rollup reporting.

The key success factor in a DSS is its ability to provide the data needed by its users, if the data record denormalization isn't right the users won't get the data they desire. A DSS system is never complete, users data requirements are always evolving over time.

DSS Tuning

Generally speaking DSS systems require tuning to allow for full table scans and range scans. The DSS system is not generally used to slice and dice data (that is the OLAP databases strength) but only for bulk rollup such as in a datamart situation. DSS systems are usually refreshed in their entirety or via bulk loads of data that correlate to specific time periods (daily, weekly, monthly, by the quarter, etc.). Indexing will usually be by dates or types of data. Data in a DSS system is generally summarized over a specific period for a specific area of a company such as monthly by division. This partitioning of data by discrete time and geographic locale leads to the ability to make full use of partition by range provided by Oracle8 as a tuning method.

DWH

A DWH, data warehouse, database is usually summarized operational data that spans the entire enterprise. The data is loaded through a clean-up and aggregation process on a predetermined interval such as daily, monthly or quarterly. One of the key concepts in data warehousing is the concept that the data is stored along a timeline. A data warehouse must support the needs of a large variety of users. A DWH may have to contain summarized, as well as atomic data. A DWH may combine the concepts of OLTP, OLAP and DSS into one physical data structure.

The major operation in a DWH is usually reporting with a low to medium level of analytical processing.

A data warehouse contains detailed, nonvolatile, time-based information. Usually data marts are derived from data warehouses. A data warehouse design should be straight forward since many users will query the data warehouse directly (however, only 10% of the queries in a DWH are usually ad-hoc in nature with 90% being canned query or reports). Data warehouse design and creation is an interative process, it is never "done". The user community must be intimately involved in the data warehouse from design through implementation or else it will fail. Generally data warehouses are denormalized structures. A normalized database stores the greatest amount of data in the smallest amount of space, in a data warehouse we sacrifice storage space for speed through denormalization.

A dyed in the wool OLTP designer may have difficulty in crossing over to the dark side of data warehousing design. Many of the time-honored concepts are bent or completely broken when designing a data warehouse. In fact, it may be impossible for a great OLTP designer to design a great DWH! Many object-related concepts can be brought to bear on a DWH design so you may find a source for DWH designers in a pool of OO developers.

DWH Tuning

DHW tuning is a complex topic. The database must be designed with a DWH multi-functional profile in mind. Tuning must be for OLTP type queries as well as bulk reporting and bulk loading operations being performed as well. Usually these tuning requirements require two or more different set of initialization parameters. One set of initialization parameters may be optimized for OLTP type operations and be used when the database is in use during normal work hours, then the database is shutdown and a new set is used for the nightly batch reporting and loading operations.

The major parameters for data warehouse tuning are:

- SHARED_POOL_SIZE – Analyze how the pool is used and size accordingly
- SHARED_POOL_RESERVED_SIZE -- ditto
- SHARED_POOL_MIN_ALLOC -- ditto
- SORT_AREA_RETAINED_SIZE – Set to reduce memory usage by non-sorting users
- SORT_AREA_SIZE – Set to avoid disk sorts if possible
- OPTIMIZER_PERCENT_PARALLEL – Set to 100% to maximize parallel processing
- HASH_JOIN_ENABLED – Set to TRUE
- HASH_AREA_SIZE – Twice the size of SORT_AREA_SIZE
- HASH_MULTIBLOCK_IO_COUNT – Increase until performance dips
- BITMAP_MERGE_AREA – If you use bitmaps alot set to 3 megabytes
- COMPATIBLE – Set to highest level for your version or new features may not be available
- CREATE_BITMAP_AREA_SIZE – During warehouse build, set as high as 12 megabytes, else set to 8 megabytes.
- DB_BLOCK_SIZE – Set only at db creation, can't be reset without rebuild, set to at least 16kb.
- DB_BLOCK_BUFFERS – Set as high as possible, but avoid swapping.
- DB_FILE_MULTIBLOCK_READ_COUNT – Set to make the value times DB_BLOCK_SIZE equal to or a multiple of the minimum disk read size on your platform, usually 64 kb or 128 kb.
- DB_FILES (and MAX_DATAFILES) – set MAX_DATAFILES as high as allowed, DB_FILES to 1024 or higher.
- DBWR_IO_SLAVES – Set to twice the number of CPUs or to twice the number of disks used for the major datafiles, whichever is less.
- OPEN_CURSORS – Set to at least 400-600
- PROCESSES – Set to at least 128 to 256 to start, increase as needed.
- RESOURCE_LIMIT – If you want to use profiles set to TRUE
- ROLLBACK_SEGMENTS – Specify to expected DML processes divided by four

- STAR_TRANSFORMATION_ENABLED – Set to TRUE if you are using star or snowflake schemas.

In addition to internals tuning, you will also need to limit the users ability to do damage by over using resources. Usually this is controlled through the use of PROFILES, later we will discuss a new feature, RESOURCE GROUPS that also helps control users. Important profile parameters are:

- SESSIONS_PER_USER – Set to maximum DOP times 4
- CPU_PER_SESSION – Determine empirically based on load
- CPU_PER_CALL -- Ditto
- IDLE_TIME – Set to whatever makes sense on your system, usually 30 (minutes)
- LOGICAL_READS_PER_CALL – See CPU_PER_SESSION
- LOGICAL_READS_PER_SESSION -- Ditto

One thing to remember about profiles is that the numerical limits they impose are not totaled across parallel sessions (except for MAX_SESSIONS).

DM

A DM or data mart is usually equivalent to a OLAP database. DM databases are specific use databases. A DM is usually created from a data warehouse for a specific division or department to use for their critical reporting needs. The data in a DM is usually summarized over a specific time period such as daily, weekly or monthly.

DM Tuning

Tuning a DM is usually tuning for reporting. You optimize a DM for large sorts and aggregations. You may also need to consider the use of partitions for a DM database to speed physical access to large data sets.

Data Warehouse Concepts

Objectives:

The objectives of this section on data warehouse Concepts are to:

1. Provide the student a grounding in data warehouse terminology
2. Provide the student with an understanding of data warehouse storage structures
3. Provide the student with an understanding of data warehouse data aggregation concepts

Data Warehouse Terminology

We have already discussed several data warehousing terms:

- DSS which stands for Decision Support System
- OLAP On-line Analytical Processing
- DM which stands for Data Mart
- Dimension – A single set of data about an item described in a fact table, a dimension is usually a denormalized table. A dimension table holds a key value and a numerical measurement or set of related measurements about the fact table object. A measurement is usually a sum but could also be an average, a mean or a variance. A dimension can have many attributes, 50 or more is the norm, since they are denormalized structures.
- Aggregate, aggregation – This refers to the process by which data is summarized over specific periods.

However, there are many more terms that you will need to be familiar with when discussing a data warehouse. Let's look at these before we go on to more advanced topics.

- Bitmap – A special form of index that equates values to bits and then stores the bits in an index. Usually smaller and faster to search than a b*tree
- Clean and Scrub – The process by which data is made ready for insertion into a data warehouse
- Cluster – A data structure in Oracle that stores the cluster key values from several tables in the same physical blocks. This makes retrieval of data from the tables much faster.
- Cluster (2) – A set of machines usually tied together with a high speed interconnect and sharing disk resources

- CUBE – CUBE enables a SELECT statement to calculate subtotals for all possible combinations of a group of dimensions. It also calculates a grand total. This is the set of information typically needed for all cross-tabular reports, so CUBE can calculate a cross-tabular report with a single SELECT statement. Like ROLLUP, CUBE is a simple extension to the GROUP BY clause, and its syntax is also easy to learn.
- Data Mining – The process of discovering data relationships that were previously unknown.
- Data Refresh – The process by which all or part of the data in the warehouse is replaced.
- Data Synchronization – Keeping data in the warehouse synchronized with source data.
- Derived data – Data that isn't sourced, but rather is derived from sourced data such as rollups or cubes
- Dimensional data warehouse – A data warehouse that makes use of the star and snowflake schema design using fact tables and dimension tables.
- Drill down – The process by which more and more detailed information is revealed
- Fact table – The central table of a star or snowflake schema. Usually the fact table is the collection of the key values from the dimension tables and the base facts of the table subject. A fact table is usually normalized.
- Granularity – This defines the level of aggregation in the data warehouse. To fine a level and your users have to do repeated additional aggregation, to course a level and the data becomes meaningless for most users.
- Legacy data – Data that is historical in nature and is usually stored offline
- MPP – Massively parallel processing – Description of a computer with many CPUs , spreads the work over many processors.
- Middleware – Software that makes the interchange of data between users and databases easier
- Mission Critical – A system that if it fails effects the viability of the company
- Parallel query – A process by which a query is broken into multiple subsets to speed execution
- Partition – The process by which a large table or index is split into multiple extents on multiple storage areas to speed processing.

- ROA – Return on Assets
- ROI – Return on investment
- Roll-up – Higher levels of aggregation
- ROLLUP -- ROLLUP enables a SELECT statement to calculate multiple levels of subtotals across a specified group of dimensions. It also calculates a grand total. ROLLUP is a simple extension to the GROUP BY clause, so its syntax is extremely easy to use. The ROLLUP extension is highly efficient, adding minimal overhead to a query.
- Snowflake – A type of data warehouse structure which uses the star structure as a base and then normalizes the associated dimension tables.
- Sparse matrix – A data structure where every intersection is not filled
- Stamp – Can be either a time stamp or a source stamp identifying when data was created or where it came from.
- Standardize – The process by which data from several sources is made to be the same.
- Star- A layout method for a schema in a data warehouse
- Summarization – The process by which data is summarized to present to DSS or DWH users.

Data Warehouse Storage Structures

Data warehouses have several basic storage structures. The structure of a warehouse will depend on how it is to be used. If a data warehouse will be used primarily for rollup and cube type operations it should be in the OLAP structure using fact and dimension tables. If a DWH is primarily used for reviewing trends, looking at standard reports and data screens then a DSS framework of denormalized tables should be used. Unfortunately many DWH projects attempt to make one structure fit all requirements when in fact many DWH projects should use a synthesis of multiple structures including OLTP, OLAP and DSS.

Many data warehouse projects use STAR and SNOWFLAKE schema designs for their basic layout. These layouts use the "FACT table -- Dimension tables" layout with the SNOWFLAKE having dimension tables that are also FACT tables.

Data warehouses consume a great deal of disk resources. Make sure you increase controllers as you increase disks to prevent IO channel saturation. Spread Oracle DWHs across as many disk resources as possible, especially with partitioned tables and indexes. Avoid RAID5 even though it offers great reliability

it is difficult if not impossible to accurately determine file placement. The excption may be with vendors such as EMC that provide high speed anticipatory caching.

Data Warehouse Aggregate Operations

The key item to data warehouse structure is the level of aggregation that the data requires. In many cases there may be multiple layers, daily, weekly, monthly, quarterly and yearly. In some cases some subset of a day may be used. The aggregates can be as simple as a summation or be averages, variances or means. The data is summarized as it is loaded so that users only have to retrieve the values. The reason the summation while loading works in a data warehouse is because the data is static in nature, therefore the aggregation doesn't change. As new data is inserted, it is summarized for its time periods not affecting existing data (unless further rollup is required for date summations such as daily into weekly, weekly in to monthly and so on.)

Data Warehouse Structure

Objectives:

The objectives of this section on data warehouse structure are to:

1. Provide the student with a grounding in schema layout for data warehouse systems
2. Discuss the benefits and problems with star, snowflake and other data warehouse schema layouts
3. Discuss the steps to build a data warehouse

Schema Structures For Data Warehousing

FLAT

A flat database layout is a fully denormalized layout similar to what one would expect in a DSS environment. All data available about a specified item is stored with it even if this introduces multiple redundancies.

Layout

The layout of a flat database is a set of tables that each reflects a given report or view of the data. There is little attempt to provide primary to secondary key relationships as each flat table is an entity unto itself.

Benefits

A flat layout generates reports very rapidly. With careful indexing a flat layout performs excellently for a single set of functions that it has been designed to fill.

Problems

The problems with a flat layout are that joins between tables are difficult and if an attempt is made to use the data in a way the design wasn't optimized for, performance is terrible and results could be questionable at best.

RELATIONAL

Tried and true but not really good for data warehouses.

Layout

The relational structure is typical OLTP layout and consists of normalized relationships using referential integrity as its cornerstone. This type of layout is typically used in some areas of a DWH and in all OLTP systems.

Benefits

The relational model is robust for many types or queries and optimizes data storage. However, for large reporting and for large aggregations performance can be brutally slow.

Problems

To retrieve data for large reports, cross-tab reports or aggregations response time can be very slow.

STAR

Twinkle twinkle...

Layout

The layout for a star structure consists of a central fact table that has multiple dimension tables that radiate out in a star pattern. The relationships are generally maintained using primary-secondary keys in Oracle and this is a requirement for using the STAR QUERY optimization in the cost based optimizer. Generally the fact tables are normalized while the dimension tables are denormalized or flat in nature. The fact table contains the constant facts about the object and the keys relating to the dimension tables while the dimension tables contain the time variant data and summations. Data warehouse and OLAP databases usually use the start or snowflake layouts.

Benefits

For specific types of queries used in data warehouses and OLAP systems the star schema layout is the most efficient.

Problems

Data loading can be quite complex.

SNOWFLAKE

As its name implies the general layout if you squint your eyes a bit, is like a snowflake.

Layout

You can consider a snowflake schema a star schema on steroids. Essentially you have fact tables that relate to dimension tables that may also be fact tables that relate to dimension tables, etc. The relationships are generally maintained using primary-secondary keys in Oracle and this is a requirement for using the STAR QUERY optimization in the cost based optimizer. Generally the fact tables are normalized while the dimension tables are denormalized or flat in nature. The fact table contains the constant facts about the object and the keys relating to the dimension tables while the dimension tables contain the time variant data and summations. Data warehouses and OLAP databases usually use the snowflake or star schemas.

Benefits

Like star queries the data in a snowflake schema can be readily accessed. The addition of the ability to add dimension tables to the ends of the star make for easier drill down into a complex data sets.

Problems

Like a star schema the data loading into a snowflake schema can be very complex.

OBJECT

The new kid on the block, but I predict big things in data warehousing for it...

Layout

An object database layout is similar to a star schema with the exception that entire star is loaded into a single object using varrays and nested tables. A snowflake is created by using REF values across multiple objects.

Benefits

Retrieval can be very fast since all data is prejoined.

Problems

Pure objects cannot be partitioned as yet, so size and efficiency are limited unless a relational/object mix is used.

Oracle and Data Warehousing

Hour 2:

Oracle7 Features

Objectives:

The objectives for this section on Oracle7 features are to:

1. Identify to the student the Oracle7 data warehouse related features
2. Discuss the limited parallel operations available in Oracle7
3. Discuss the use of partitioned views
4. Discuss multi-threaded server and its application to the data warehouse
5. Discuss high-speed loading techniques available in Oracle7

Oracle7 Data Warehouse related Features

Use of Partitioned Views

In late Oracle7 releases the concept of partitioned views was introduced. A partitioned view consists of several tables, identical except for name, joined through a view. A partition view is a view that for performance reasons brings together several tables to behave as one.

The effect is as though a single table were divided into multiple tables (partitions) that could be independently accessed. Each partition contains some subset of the values in the view, typically a range of values in some column. Among the advantages of partition views are the following:

- Each table in the view is separately indexed, and all indexes can be scanned in parallel.
- If Oracle can tell by the definition of a partition that it can produce no rows to satisfy a query,
- Oracle will save time by not examining that partition.
- The partitions can be as sophisticated as can be expressed in CHECK constraints.
- If you have the parallel query option, the partitions can be scanned in parallel.
- Partitions can overlap.
- Among the disadvantages of partition views are the following:
- They (the actual view) cannot be updated. The underlying tables however, can be updated.
- They have no master index; rather each component table is separately indexed. For this reason, they are recommended for DSS (Decision Support Systems or "data warehousing") applications, but not for OLTP.

To create a partition view, do the following:

1. CREATE the tables that will comprise the view or ALTER existing tables suitably.
2. Give each table a constraint that limits the values it can hold to the range or other restriction criteria desired.
3. Create a local index on the constrained column(s) of each table.
4. Create the partition view as a series of SELECT statements whose outputs are combined using UNION ALL. The view should select all rows and columns from the underlying tables. For more information on SELECT or UNION ALL, see "SELECT" .
5. If you have the parallel query option enabled, specify that the view is parallel, so that the tables within it are accessed simultaneously when the view is queried. There are two ways to do this:

    - specify "parallel" for each underlying table.
    - place a comment in the SELECT statement that the view contains to give a hint of "parallel" to the Oracle optimizer.

There is no special syntax required for partition views. Oracle interprets a UNION ALL view of several tables, each of which have local indexes on the same columns, as a partition view. To confirm that Oracle has correctly identified a partition view, examine the output of the EXPLAIN PLAN command.

In releases prior to 7.3 use of partition views was frowned upon since the optimizer was not able to be partition aware thus for most queries all of the underlying tables where searched rather than just the affected tables. After 7.3 the optimizer became more parition biew friendly and this is no longer the case.

An example query to build a partition view would be:

```
CREATE OR REPALCE VIEW acct_payable AS
SELECT * FROM acct_pay_jan99
UNION_ALL
SELECT * FROM acct_pay_feb99
UNION_ALL
SELECT * FROM acct_pay_mar99
UNION_ALL
SELECT * FROM acct_pay_apr99
UNION_ALL
SELECT * FROM acct_pay_may99
UNION_ALL
SELECT * FROM acct_pay_jun99
UNION_ALL
SELECT * FROM acct_pay_jul99
UNION_ALL
SELECT * FROM acct_pay_aug99
UNION_ALL
SELECT * FROM acct_pay_sep99
UNION_ALL
SELECT * FROM acct_pay_oct99
UNION_ALL
SELECT * FROM acct_pay_nov99
UNION_ALL
SELECT * FROM acct_pay_dec99;
```

A select from the view using a range such as:

```
SELECT * FROM account_payables
WHERE payment_date BETWEEN '1-jan-1999' AND '1-mar-1999';
```

Would be resolved by querying the table acct_pay_jan99 and acct_pay_feb99 only in versions after 7.3. Of course if you are in Oracle8 true partitioned tables should be used instead.

Use of Oracle Parallel Query Option

The Parallel Query Option (PQ)) should not be confused with the shared database or parallel database option (Oracle parallel server – OPS). Parallel

query server relates to parallel query and DDL operations while parallel or shared database relates to multiple instances using the same central database files.

Due to the size of tables in most data warehouses, the datafiles in which the tables reside cannot be made large enough to hold a complete table. This means multiple datafiles must be used, sometimes, multiple disks as well. These huge file sized result in extremely lengthy query times if the queries are perfomed in serial. Oracle provides for the oracle parallel query option to allow this multi-datafile or multi-disk configuration to work for, instead of against you. In parallel query a query is broken into multiple sub-queries that are issued to as many query processes as are configured and available. The net effect of increasing the number of query servers acting on your befalf in a query is that while the serial time remains the same, the time is broken into multiple simultaneous intervals thus reducing the overall time spent doing a large query or operation to X/N where X is the original time in serial mode and N is the number of query processes. In reality the time is always greater than X/N because of processing overhead after the query slave processes return the results and the query slaves responsible for sorting and grouping do their work.

The use of parallel table and index builds also speeds data loading but remember that N extents will be created where N is equal to the number of slaves acting on the build request and each will require an initial extent to work in temporarily. Once the slaves complete the work on each table or index extent the extents are merged and any unused space is returned to the tablespace.

To use parallel query the table must be created or altered to have a degree of parallel set. In Oracle7 the syntax for the parallel option on a table creation is:

```
CREATE TABLE table_name (column_list)
Storage_options
Table_optins
NOPARALLEL|PARALLEL(DEGREE n|DEFAULT)
```

If a table is created as parallel or is altered to be parallel then any index created on that table will be created in parallel even though the index will not itself be a parallel capable index.

If default is specified for the degree of parallel, the value for DEGREE will be determined from the number of CPUs and/or the number of devices holding the table's extents.

Oracle7 Parallel Query Server is configured using the following initialization parameters:

- PARALLEL_MIN_SERVERS – Sets the minimum number of parallel servers, can never go below this level inspite of exceeding PARALLEL_SERVER_IDLE_TIME
- PARALLEL_SERVER_IDLE_TIME – If a server is idle this long it is killed
- PARALLEL_MAX_SERVERS – Maximum number of servers that can be started, will shrink back to PARALLEL_MIN_SERVERS.

Use of MTS

MTS, or multi-threaded server, is really intended for systems where there are a large number of users (over 150) and a limited amount of memory. The multi-threaded serve is set up using the following initialization parameters:

- SHARED_POOL_SIZE – needs to be increased to allow for UGA
- MTS_LISTENER_ADDRESS – Sets the address for the listener
- MTS_SERVICE – Names the service (usually the same as SID)
- MTS_DISPATCHERS – Sets the base number of dispatchers
- MTS_MAX_DISPATCHERS – Sets the maximum number of dispatchers
- MTS_SERVERS – Sets the minimum number of servers
- MTS_MAX_SERVERS – Sets the maximum number of servers

If you have a low number of users and no memory problems, using MTS can reduce your performance. MTS is most useful in an OLTP environment where a large number of users may sign on to the database but only a few are actually doing any work concurrently.

Oracle8 Features

Objectives:

The objectives for this section on Oracle8 features are to:

1. Identify to the student the Oracle8 data warehouse related features
2. Discuss the use of partitioned tables and indexes
3. Discuss the expanded parallel abilities of Oracle8
4. Discuss the star query/structure aware capabilities of the optimizer

5. Discuss new indexing options
6. Discuss new Oracle8 internals options
7. Discuss RMAN and its benefits in Oracle8 for data warehousing

Partitioned Tables and Indexes

In Oracle7 we discussed the use of partitioned views. Partitioned views had several problems. First, each table in a partitioned view was maintained separately. Next, the indexes where independent for each table in a partitioned view. Finally, some operations still weren't very efficient on a partitioned view. In Oracle8 we have true table and index partitioning where the system maintains range partitioning, maintains indexes and all operations are supported against the partitioned tables. Partitions are good because:

- Each partition is treated logically as its own object. It can be dropped, split or taken offine without affecting other partitions in the same object.
- Rows inside partitions can be managed separately from rows in other partitions in the same object. This is supported by the extended partition syntax.
- Maintenance can be performed on individual partitions in an object, this is all known as partiion independence.
- Storage values (initial, necxt, ext) can be different between individual partitions or can be inherited.
- Partitions can be loaded without affecting other partitions.

Instead of creating several tables and then using a view to trick Oracle into treating them as a single table we create a single table and let Oracle do the work to maintain it as a partitioned table. A partitioned table in Oracle8 is range partitioned, for example on month, day, year or some other integer or numeric value. This makes partitioning of tables ideal for the time-based data that is the main-stay of data warehousing.

So our accounts payable example from the partitioned view section would become:

```
CREATE TABLE acct_pay_99 (acct_no NUMBER, acct_bill_amt NUMBER, bill_date
DATE, paid_date DATE, penalty_amount NUMBER, chk_number NUMBER)
STORAGE (INITIAL 40K NEXT 40K PCTINCREASE 0)
PARTITION BY RANGE (paid_date)
(
PARTITION acct_pay_jan99
   VALUES LESS THAN (TO_DATE('01-feb-1999','DD-mon-YYYY'))
```

```
   TABLESPACE acct_pay1,
PARTITION acct_pay_feb99
   VALUES LESS THAN (TO_DATE('01-mar-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_mar99
   VALUES LESS THAN (TO_DATE('01-apr-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_apr99
   VALUES LESS THAN (TO_DATE('01-may-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_may99
   VALUES LESS THAN (TO_DATE('01-jun-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_jun99
   VALUES LESS THAN (TO_DATE('01-jul-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_jul99
   VALUES LESS THAN (TO_DATE('01-aug-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_aug99
   VALUES LESS THAN (TO_DATE('01-sep-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_sep99
   VALUES LESS THAN (TO_DATE('01-oct-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_oct99
   VALUES LESS THAN (TO_DATE('01-nov-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay1,
PARTITION acct_pay_nov99
   VALUES LESS THAN (TO_DATE('01-dec-1999','DD-mon-YYYY'))
   TABLESPACE acct_pay11,
PARTITION acct_pay_dec99
   VALUES LESS THAN (TO_DATE('01-jan-2000','DD-mon-YYYY'))
   TABLESPACE acct_pay12,
PARTITION acct_pay_2000
   VALUES LESS THAN (MAXVALUE))
  TABLESPACE acct_pay_max
/
```

The above command results in a partitioned table that can be treated as a single table for all inserts, updates and deletes or, if desired, the individual partitions can be addressed. In addition the indexes created will be by default local indexes that are automatically partitioned the same way as the base table. Be sure to specify tablespaces for the index partitions or they will be placed with the table partitions.

In the example the paid_date is the partition key which can have up to 16 columns included. Deciding the partition key can be the most vital aspect of creating a successful data warehouse using partitions. I suggest using the UTLSIDX.SQL script series to determine the best combination of key values. The UTLSIDX.SQL script series is documented in the script headers for UTLSIDX.SQL, UTLOIDXS.SQL and UTLDIDXS.SQL script SQL files. Essentially you want to determine how many key values or concatenated key

values there will be and how many rows will correspond to each key value set. In many cases it will be important to balance rows in each partition so that IO is balanced. However in other cases you may want hard separation based on the data ranges and you don't really care about the number of records in each partition, this needs to be determined on a warehouse-by-warehouse basis.

Oracle8 Enhanced Parallel DML

To use parallel anything in Oracle8 the parallel server parameters must be set properly in the initialization file, these parameters are:

- COMPATIBLE Set this to at least 8.0
- CPU_COUNT this should be set to the number of CPUs on your server, if it isn't set it manually.
- DML_LOCKS set to 200 as a start for a parallel system.
- ENQUEUE_RESOURCES set this to DML_LOCKS+20
- OPTIMIZER_PERCENT_PARALLEL this defaults to 0 favoring serial plans, set to 100 to force all possible parallel operations or somewhere in between to be on the fence.
- PARALLEL_MIN_SERVERS set to the minimum number of parallel server slaves to start up.
- PARALLEL_MAX_SERVERS set to the maximum number of parallel slaves to start, twice the number of CPUs times the number of concurrent users is a good beginning.
- SHARED_POOL_SIZE set to at least ((3*msgbuffer_size)*(CPUs*2)*PARALLEL_MAX_SERVERS) bytes + 40 megabytes
- ALWAYS_ANTI_JOIN Set this to HASH or NOT IN operations will be serial.
- SORT_DIRECT_WRITES Set this to AUTO

DML, data manipulation language, what we know as INSERT, UPDATE and DELETE as well as SELECT can use parallel processing, the list of parallel operations supported in Oracle8 is:

- Table scan
- NOT IN processing
- GROUP BY processing

- SELECT DISTINCT
- AGGREGATION
- ORDER BY
- CREATE TABLE x AS SELECT FROM y;
- INDEX maintenance
- INSERT INTO x ... SELECT ... FROM y
- Enabling constraints (index builds)
- Star transformation

In some of the above operations the table has to be partitioned to take full advantage of the parallel capability. In some releases of Oracle8 you have to explicitly turn on parallel DML using the ALTER SESSION command:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Remember that the COMPATIBLE parameter must be set to at least 8.0.0 to get parallel DML. Also, parallel anything doesn't make sense if all you have is one CPU. Make sure that your CPU_COUNT variable is set correctly, this should be automatic but problems have been reported on some platforms.

Oracle8 supports parallel inserts, updates, and deletes into partitioned tables. It also supports parallel inserts into non-partitioned tables. The parallel insert operation on a non-partitioned table is similar to the direct path load operation that is available in Oracle7. It improves performance by formatting and writing disk blocks directly into the datafiles, bypassing the buffer cache and space management bottlenecks. In this case, each parallel insert process inserts data into a segment above the high watermark of the table. After the transaction commits, the high watermark is moved beyond the new segments.

To use parallel DML, it must be enabled prior to execution of the insert, update, or delete operation. Normally, parallel DML operations are done in batch programs or within an application that executes a bulk insert, update, or delete. New hints are available to specify the parallelism of DML statements.

I suggest using explain plan and tkprof to verify that operations you suspect are parallel are actually parallel. If you find for some reason Oracle isn't doing parallel processing for an operation which you feel should be parallel, use the parallel hints to force parallel processing:

- PARALLEL
- NOPARALLEL

- APPEND
- NOAPPEND
- PARALLEL_INDEX

An example would be:

```
SELECT /*+ FULL(clients) PARALLEL(clients,5,3)*/ client_id, client_name,
client_address FROM clients;
```

By using hints the developer and tuning DBA can exercise a high level of control over how a statement is processed using the parallel query option.

Oracle8 Enhanced Optimizer Features

The Optimizer in Oracle8 has been dramatically improved to recognize and utilize partitions, to use new join and anti-join techniques and in general to do a better job of tuning statements.

Oracle8 introduces performance improvements to the processing of star queries, which are common in data warehouse applications. Oracle7 introduced the functionality of star query optimization, which provides performance improvements for these types of queries. In Oracle8, star-query processing has been improved to provide better optimization for star queries.

In Oracle8, a new method for executing star queries was introduced. Using a more efficient algorithm, and utilizing bitmapped indexes, the new star-query processing provided a significant performance boost to data warehouse applications.

Oracle8 has superior performance with several types of star queries, including star schemas with "sparse" fact tables where the criteria eliminate a great number of the fact table rows. Also, when a schema has multiple fact tables, the optimizer efficiently processes the query. Finally, Oracle8 can efficiently process star queries with large or many dimension tables, unconstrained dimension tables, and dimension tables that have a "snowflake" schema design.

Oracle8's star-query optimization algorithm, unlike that of Oracle7, does not produce any Cartesian-product joins. Star queries are now processed in two basic phases. First, Oracle8 retrieves only the necessary rows from the fact table. This retrieval is done using bit mapped indexes and is very efficient. The second phase joins this result set from the fact table to the relevant dimension tables. This allows for better optimizations of more complex star queries, such as those with multiple fact tables. The new algorithm uses bit-mapped indexes, which offer significant storage savings over previous methods that required

concatenated column B-tree indexes. The new algorithm is also completely parallelized, including parallel index scans on both partitioned and non-partitioned tables.

Oracle8 Enhanced Index Structures

Oracle8 provides enhancements to the bitmapped indexes introduced in Oracle7. Also, a new feature know as index-only tables or IOTs was introduced to allow tables where the entire key is routinely retrived to be stored in a more efficient B*tree structure with no need for supporting indexes.

Also introduced in Oracle8 is the concept of reverse key indexes. When large quantities of data are loaded using a key value derived from either SYSDATE or from sequences unbalancing of the resulting index B*tree can result. Reverse key indexes reduce the "hot spots" in indexes, especially ascending indexes. Unbalanced indexes can cause the index to become increasingly deep as the base table grows. Reverse key indexes reverse the bytes of leaf-block entries, therefore preventing "sliding indexes".

Oracle8 Enhanced Internals Features

In Oracle8 you can have multiple DBWR (up to 10) processes as well as database writer slave processes. Also added is the ability to have multiple log writer slaves.

The memory structures have also been altered in Oracle8. Oracle has added the ability to have multiple buffer pools. In Oracle7 all data was kept in a single buffer pool and was subject to aging of the LRU algorithm as well as flushing caused by large full table scans. In a data warehouse environment it was difficult to get hit ratios above 60-70% for the buffer pool. Now in Oracle8 you have two additional buffer pools that can be used to sub-divide the default buffer pool. The two new buffer pools are the KEEP and RECYCLE pools. The KEEP sub-pool is used for those objects such as reference tables that you want kept in the pool. The RECYCLE pool is used for large objects that are accessed piece-wise such as LOB objects or partitioned objects. Items such as tables or indexes are assigned to the KEEP or RECYCLE pools when they are created or can be altered to use the new pools. Multiple database writers and LRU latches are configured to maintain the new pools.

Another new memory structure in Oracle8 is the large pool. The large pool is used to relieve the shared pool from UGA duties when MTS is used. The large pool also keeps the recovery and backup process IO queues. By configuring the large pool in a data warehouse you can reduce the thrashing of the shared pool and improve backup and recovery response as well as improve MTS and PQO response. In fact if PQO is initialized the large pool is automatically configured.

Backup and Recovery Using RMAN

In Oracle7 oracle gave us Enterprise Backup (EBU) unfortunately it was difficult to use and didn't give us any additional functionality over other backup tools, at least not enough to differentiate it. In Oracle8 we now have the Recovery Manager (RMAN) product. The RMAN product replaces EBU and provides expanded capabilities such as tablespace point-in-time recovery and incremental backups.

Of primary importance in data warehousing is the speed and size of the required backups. Using Oracle8's RMAN facility only the changed blocks are written out to a backup set using the incremental feature. This process of only writing changed blocks substantially reduces the size of backups and thus the time required to create a backup set. RMAN also provides a catalog feature to track all backups and automatically tell you through requested reports when a file needs to be backed up and what files have been backed up.

Data Warehousing 201

Hour 1:

Oracle8i Features

Objectives:

The objectives for this section on Oracle8i features are to:

1. Discuss SQL options applicable to data warehousing
2. Discuss new partitioning options in Oracle8i
3. Show how new user-defined statistics are used for Oracle8i tuning
4. Discuss dimensions and hierarchies in relation to materialized views and query rewrite
5. Discuss locally managed tablespaces and their use in data warehouses
6. Discuss advanced resource management through plans and groups.
7. Discuss the use of row level security and data warehousing

Oracle8i SQL Enhancements for Data Warehouses

Oracle8i has provided many new features for use in a data warehouse environment that make tuning of SQL statements easier. Specifically, new SQL operators have been added to significantly reduce the complexity of SQL statements that are used to perform cross-tab reports and summaries. The new SQL operators that have been added for use with SELECT are the CUBE and ROLLUP operators. Another operator is the SAMPLE clause which allows the user to specify random sampling of rows or blocks. The SAMPLE operator is useful for some data mining techniques and can be used to avoid full table scans.

There are also several new indexing options available in Oracle8i, function based indexes, descending indexes and enhancements to bitmapped indexes are provided.

Function Based Indexes

Function based indexes as their name implies are indexes based on functions. In previous releases of Oracle if we wanted to have a column that was always searched uppercase (for example a last name that could have mixed case like McClellum) we had to place the returned value with its mixed case letters in one column and add a second column that was upper-cased to index and use in searches. This doubling of columns required for this type of searching lead to doubling of size requirements for some application fields. The cases where more complex such as SOUNDEX and other functions would also have required use of a second column. This is not the case with Oracle8i, now functions and user-defined functions as well as methods can be used in indexes. Let's look at a simple example using the UPPER function.

```
CREATE INDEX tele_dba.up1_clientsv81
ON tele_dba.clientsv81(UPPER(customer_name))
TABLESPACE tele_index
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0);
```

In many applications a column may store a numeric value that translates to a minimal set of text values, for example a user code that designates functions such as 'Manager', 'Clerk', or 'General User'. In previous versions of Oracle you would have had to perform a join between a lookup table and the main table to search for all 'Manager' records. With function indexes the DECODE function can be used to eliminate this type of join.

```
CREATE INDEX tele_dba.dec_clientsv81
ON tele_dba.clientsv81(DECODE(user_code,
1,'MANAGER',2,'CLERK',3,'GENERAL USER'))
TABLESPACE tele_index
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0);
```

A query against the clientsv8i table that would use the above index would look like:

```
SELECT customer_name FROM tele_dba.clientsv8i
WHERE DECODE(user_code,
1,'MANAGER',2,'CLERK',3,'GENERAL USER')='MANAGER';
```

The explain plan for the above query shows that the index will be used to execute the query:

```
SQL> SET AUTOTRACE ON EXPLAIN
SQL> SELECT customer_name FROM tele_dba.clientsv8i
  2  WHERE DECODE(user_code,
  3* 1,'MANAGER',2,'CLERK',3,'GENERAL USER') = 'MANAGER'

no rows selected


Execution Plan
----------------------------------------------------------------------
0   SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=526)
1 0 TABLE ACCESS (BY INDEX ROWID) OF 'CLIENTSV8I' (Cost=1 Card=1
Bytes=526)
2 1 INDEX (RANGE SCAN) OF 'DEC_CLIENTSV8I' (NON-UNIQUE) (Cost=1 Card=1)
```

The table using function based indexes must be analyzed and the optimizer mode set to CHOOSE or the function based indexes will not be used. In addition, just like materialized views, the QUERY_REWRITE_ENABLED and QUERY_REWRITE_INTEGRITY initialization parameters must be set, or they must be set using the ALTER SESSION command in order for function based indexes to be utilized in query processing. The RULE based optimizer cannot use function based indexes.

If the function based index is built using a user defined function, any alteration or invalidation of the user function will invalidate the index. Any user built functions must not contain aggregate functions and must be deterministic in nature. A deterministic function is one that is built using the DETERMINISTIC key word in the CREATE FUNCTION, CREATE PACKAGE or CREATE TYPE commands. A deterministic function is defined as one that always returns the same set value given the same input no matter where the function is executed from within your application. As of 8.1.5 the validity of the DETERMINISTIC key word usage is not verified and it is left up to the programmer to ensure it is used properly. A function based index cannot be created on a LOB, REF or nested table column or against an object type that contains a LOB, REF or nested table. Let's look at an example of a user defined type (UDT) method.

```
CREATE TYPE room_t AS OBJECT(
lngth NUMBER,
width NUMBER,
MEMBER FUNCTION SQUARE_FOOT
RETURN NUMBER DETERMINISTIC);
/
CREATE TYPE BODY room_t AS
  MEMBER FUNCTION SQUARE_FOOT
  RETURN NUMBER IS
  area NUMBER;
  BEGIN
   AREA:=lngth*width;
    RETURN area
  END;
END;
```

```
/
CREATE TABLE rooms OF room_t
TABLESPACE test_data
STORAGE (INITIAL 100K NEXT 100K PCTINCREASE 0);

CREATE INDEX area_idx ON rooms r (r.square_foot());
```

Note: the above example is based on the examples given in the oracle manuals, when tested on 8.1.3 the DETERMINISTIC keyword caused an error, dropping the DETERMINISTIC keyword allowed the type to be created, however, the attempted index creation failed on the alias specification. In 8.1.3 the key word is REPEATABLE instead of DETERMINISTIC, however, even when specified with the REPEATABLE keyword the attempt to create the index failed on the alias.

A function based index is allowed to be either a normal B*tree index or it can also be mapped into a bitmapped format.

Reverse Key Index

A reversed key index prevents unbalancing of the b*-tree and the resulting hot blocking which will happen if the b*-tree becomes unbalanced. Generally, unbalanced b*trees are caused by high volume insert activity in a parallel server where the key value is only slowly changing such as with an integer generated from a sequence or a data value. A reverse key index works by reversing the order of the bytes in the key value, of course the ROWID value is not altered, just the key value. The only way to create a reverse key index is to use the CREATE INDEX command, an index that is not reverse key cannot be altered or rebuilt into a reverse key index, however, a reverse key index can be rebuilt to be a normal index.

One of the major limitations of reverse key indexes are that they cannot be used in an index range scan since reversing the index key value randomly distributes the blocks across the index leaf nodes. A reverse key index can only use the fetch-by-key or full-index(table)scans methods of access. Let's look at an example.

```
CREATE INDEX rpk_po ON tele_dba.po(po_num) REVERSE
TABLESPACE tele_index
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0);
```

The above index would reverse the values for the po_num column is it creates the index. This would assure random distribution of the values across the index leaf-nodes. But what if we then determine that the benefits of the reverse key do not out weigh the draw backs? We can use the ALTER command to rebuild the index as a noreverse index:

```
ALTER INDEX rpk_po REBUILD NOREVERSE;
ALTER INDEX rpk_po RENAME TO pk_po;
```

While the manuals only discuss the benefits of the reverse key index in the realm of Oracle Parallel Server, if you experience performance problems after a bulk load of data, dropping and recreating the indexes involved as reverse key indexes may help if the table will continue to be loaded in a bulk fashion.

Bitmapped Index Improvements

The improvements to bitmapped indexes enhance query performance. One enhancement lifts the restriction where by a bitmapped index was invalidated if its table was altered. Two new clauses where added to the ALTER TABLE command that directly affect bitmapped indexes:

- MINIMIZE RECORDS PER BLOCK
- NOMINIMIZE RECORDS PER BLOCK

These options permit tuning of the ROWID-to-Bitmap mapping. The MINIMIZE option is used to optimize bitmap indexes for a query-only environment by requesting that the most efficient possible mapping of bits to ROWIDs. The NOMINIMIZE option turns this feature off.

Use of Hints

Another new SQL feature is the ability to use hints to force parallelization of aggregate distinct queries even if they don't contain a GROUP BY clause. These hints are:

- PARALLEL
- PQ_DISTRIBUTE
- PARALLEL_INDEX

Oracle8i Data Warehouse Table Options

There are several enhancements to the table concept in Oracle8i, particularly in the area of partitioned tables.

Partitioned Table Enhancements

A partitioned table has to be a straight relational table in Oracle8, in Oracle8i this restriction is removed and you must be careful to allow for all LOB or Nested

storage to be carried through to all partition storage areas. A partitioned table is used to split up a table's data into separate physical as well as logical areas. This gives the benefits of being able to break up a large table in more manageable pieces and allows the Oracle8 kernel to more optimally retrieve values. Let's look at a quick example. We have a sales entity that will store results from sales for the last twelve months. This type of table is a logical candidate for partitioning because:

1. Its values have a clear separator (months).
2. It has a sliding range (the last year).
3. We usually access this type of date by sections (months, quarters, years).

The DDL for this type of table would look like this:

```
CREATE TABLE sales (
acct_no               NUMBER(5),
sales_person          VARCHAR2(32),
sales_month           NUMBER(2),
amount_of_sale        NUMBER(9,2),
po_number             VARCHAR2(10))
PARTITION BY RANGE (sales_month)
     PARTITION sales_mon_1 VALUES LESS THAN (2),
     PARTITION sales_mon_2 VALUES LESS THAN (3),
     PARTITION sales_mon_3 VALUES LESS THAN (4),
          ...
     PARTITION sales_mon_12 VALUES LESS THAN (13),
     PARTITION sales_bad_mon VALUES LESS THAN (MAXVALUE));
```

In the above example we created the sales table with 13 partitions, one for each month plus an extra to hold improperly entered months (values >12). Always specify a last partition to hold MAXVALUE values for your partition values.

Using Subpartitioning

New to Oracle8i is the concept of subpartitioning. This subpartitioning allows a table partition to be further subdivided to allow for better spread of large tables. In this example we create a table for tracking the storage of data items stored by various departments. We partition by storage date on a quarterly basis and do a further storage subpartition on data_item. The normal activity quarters have 4 partitions, the slowest has 2 and the busiest has 8.

```
CREATE TABLE test5 (data_item INTEGER, length_of_item INTEGER,
                    storage_type VARCHAR(30),
                    owning_dept NUMBER, storage_date DATE)
   PARTITION BY RANGE (storage_date)
   SUBPARTITION BY HASH(data_item)
   SUBPARTITIONS 4
```

```
   STORE IN (data_tbs1, data_tbs2,
             data_tbs3, data_tbs4)
   (PARTITION q1_1999
      VALUES LESS THAN (TO_DATE('01-apr-1999', 'dd-mon-yyyy')),
    PARTITION q2_1999
      VALUES LESS THAN (TO_DATE('01-jul-1999', 'dd-mon-yyyy')),
    PARTITION q3_1999
      VALUES LESS THAN (TO_DATE('01-oct-1999', 'dd-mon-yyyy'))
      (SUBPARTITION q3_1999_s1 TABLESPACE data_tbs1,
       SUBPARTITION q3_1999_s2 TABLESPACE data_tbs2),
    PARTITION q4_1999
      VALUES LESS THAN (TO_DATE('01-jan-2000', 'dd-mon-yyyy'))
      SUBPARTITIONS 8
      STORE IN (q4_tbs1, q4_tbs2, q4_tbs3, q4_tbs4,
                q4_tbs5, q4_tbs6, q4_tbs7, q4_tbs8),
    PARTITION q1_2000
      VALUES LESS THAN (TO_DATE('01-apr-2000', 'dd-mon-yyyy'))):
/
```

The items to notice in the above code example is that the partition level commands override the default subpartitioning commands, thus, partition Q3_1999 only gets two subpartitions instead of the default of 4 and partition Q4_1999 gets 8. The main partitions are partitioned based on date logic while the subpartitions use a hash value calculated off of a varchar2 value. The subpartitioning is done on a round robin fashion depending on the hash value calculated filling the subpartitions equally.
Note that no storage parameters where specified in the example, I created the tablespaces such that the default storage for the tablespaces matched what I needed for the subpartitions. This made the example code easier to write and clearer to use for the visualization of the process involved.

Using Oracle8i Temporary Tables

Temporary tables are a new feature of Oracle8i. There are two types of temporary tables, GLOBAL TEMPORARY and TEMPORARY. A GLOBAL TEMPORARY table is one whose data is visible to all sessions, a TEMPORARY table has contents only visible to the session that is using it. In version 8.1.3 the TEMPORARY key word could not be specified without the GLOBAL modifier. In addition, a temporary table can have session-specific or transaction specific data depending on how the ON COMMIT clause is used in the tables definition. The temporary table doesn't go away when the session or sessions are finished with it; however, the data in the table is removed. Here is an example creation of both a preserved and deleted temporary table:

```
SQL> CREATE TEMPORARY TABLE test6 (
  2     starttestdate DATE,
  3     endtestdate DATE,
  4     results NUMBER)
  5  ON COMMIT DELETE ROWS
  6  /
CREATE TEMPORARY TABLE test6 (
```

```
       *
ERROR at line 1:
ORA-14459: missing GLOBAL keyword

SQL>   CREATE GLOBAL TEMPORARY TABLE test6 (
  2     starttestdate DATE,
  3     endtestdate DATE,
  4     results NUMBER)
  5* ON COMMIT PRESERVE ROWS
SQL> /

Table created.

SQL> desc test6
 Name                           Null?    Type
 ------------------------------ -------- ----
 STARTTESTDATE                           DATE
 ENDTESTDATE                             DATE
 RESULTS                                 NUMBER


SQL> CREATE GLOBAL TEMPORARY TABLE test7 (
  2     starttestdate DATE,
  3     endtestdate DATE,
  4     results NUMBER)
  5  ON COMMIT DELETE ROWS
  6  /

Table created.

SQL> desc test7
 Name                           Null?    Type
 ------------------------------ -------- ----
 STARTTESTDATE                           DATE
 ENDTESTDATE                             DATE
 RESULTS                                 NUMBER


SQL> insert into test6 values (sysdate, sysdate+1, 100);

1 row created.

SQL> commit;

Commit complete.

SQL> insert into test7 values (sysdate, sysdate+1, 100);

1 row created.

SQL> select * from test7;

STARTTEST ENDTESTDA    RESULTS
--------- --------- ----------
29-MAR-99 30-MAR-99        100

SQL> commit;
```

```
Commit complete.

SQL> select * from test6;

STARTTEST ENDTESTDA    RESULTS
--------- --------- ----------
29-MAR-99 30-MAR-99        100

SQL> select * from test7;

no rows selected

SQL>
```

The items to notice in this example are that I had to use the full GLOBAL TEMPORARY specification (on 8.1.3), I received a syntax error when In tried to create a session specific temporary table. Next, notice that with the PRESERVE option the commit resulting in the retention of the data, while with the DELETE option, when the transaction committed the data was removed from the table. When the session was exited and then re-entered the data had been removed from the temporary table. Even with the GLOBAL option set and select permission granted to public on the temporary table I couldn't see the data in the table from another session. I could however perform a describe the table and insert my own values into it, which then the owner couldn't select.

Creation Of An Index Only Table

Index only tables have been around since Oracle8.0. If neither the HASH or INDEX ORGANIZED options are used with the create table command then a table is created as a standard hash table. If the INDEX ORGANIZED option is specified, the table is created as a B-tree organized table identical to a standard Oracle index created on similar columns. Index organized tables do not have rowids.

Index organized tables have the option of allowing overflow storage of values that exceed optimal index row size as well as allowing compression to be used to reduce storage requirements. Overflow parameters can include columns to overflow as well as the percent threshold value to begin overflow. An index organized table must have a primary key. Index organized tables are best suited for use with queries based on primary key values. Index organized tables can be partitioned in Oracle8i as long as they do not contain LOB or nested table types. The pcthreshold value specifies the amount of space reserved in an index block for row data, if the row data length exceeds this value then the row(s) are stored in the area specified by the OVERFLOW clause. If no overflow clause is specified rows that are too long are rejected. The INCLUDING COLUMN clause allows you to specify at which column to break the record if an overflow occurs. For example:

```
CREATE TABLE test8
  ( doc_code CHAR(5),
    doc_type INTEGER,
    doc_desc VARCHAR(512),
       CONSTRAINT pk_docindex PRIMARY KEY (doc_code,doc_type) )
  ORGANIZATION INDEX TABLESPACE data_tbs1
  PCTTHRESHOLD 20 INCLUDING doc_type
  OVERFLOW TABLESPACE data_tbs2
/
```

In the above example the IOT **test8** has three columns, the first two of which make up the key value. The third column in **test8** is a description column containing variable length text. The PCTHRESHOLD is set at 20 and if the threshold is reached the overflow goes into an overflow storage in the **data_tbs2** tablespace with any values of **doc_desc** that won't fit in the index block. Note that you will the best performance from IOTs when the complete value is stored in the IOT structure, otherwise you end up with an index and table lookup as you would with a standard index-table setup.

Oracle8i and Tuning of Data Warehouses using Small Test Databases

In previous releases of Oracle in order to properly tune a database or data warehouse you had to have data that was representative of the volume expected or results where not accurate. In Oracle8i the developer and DBA can either export statistics from a large production database or simply add them themselves to make the optimizer think the tables are larger than they are in your test database. The Oracle provided package DBMS_STATS provides the mechanism by which statistics are manipulated in the Oracle8i database. This package provides a mechanism for users to view and modify optimizer statistics gathered for database objects. The statistics can reside in two different locations:

- in the dictionary
- in a table created in the user's schema for this purpose

Only statistics stored in the dictionary itself will have an impact on the cost-based optimizer.

This package also facilitates the gathering of some statistics in parallel.

The package is divided into three main sections:

- procedures which set/get individual stats.
- procedures which transfer stats between the dictionary and user stat tables.

- procedures which gather certain classes of optimizer statistics and have improved (or equivalent) performance characteristics as compared to the analyze command.

Most of the procedures include the three parameters: statown, stattab, and statid. These parameters are provided to allow users to store statistics in their own tables (outside of the dictionary) which will not affect the optimizer. Users can thereby maintain and experiment with "sets" of statistics without fear of permanently changing good dictionary statistics. The stattab parameter is used to specify the name of a table in which to hold statistics and is assumed to reside in the same schema as the object for which statistics are collected (unless the statown parameter is specified). Users may create multiple such tables with different stattab identifiers to hold separate sets of statistics. Additionally, users can maintain different sets of statistics within a single stattab by making use of the statid parameter (which can help avoid cluttering the user's schema).

For all of the set/get procedures, if stattab is not provided (i.e., null), the operation will work directly on the dictionary statistics; therefore, users need not create these statistics tables if they only plan to modify the dictionary directly. However, if stattab is not null, then the set/get operation will work on the specified user statistics table, not the dictionary.
This package provides a mechanism for users to view and modify optimizer statistics gathered for database objects. The statistics can reside in two different locations:

- in the dictionary
- in a table created in the user's schema for this purpose

Only statistics stored in the dictionary itself will have an impact on the cost-based optimizer.

This package also facilitates the gathering of some statistics in parallel.

The package is divided into three main sections:

- procedures which set/get individual stats.
- procedures which transfer stats between the dictionary and user statistics tables.
- procedures which gather certain classes of optimizer statistics and have improved (or equivalent) performance characteristics as compared to the analyze command.

Most of the procedures include the three parameters: statown, stattab, and statid. These parameters are provided to allow users to store statistics in their own tables (outside of the dictionary) which will not affect the optimizer. Users can thereby maintain and experiment with "sets" of statistics without fear of permanently changing good dictionary statistics. The stattab parameter is used to specify the name of a table in which to hold statistics and is assumed to reside in the same schema as the object for which statistics are collected (unless the statown parameter is specified). Users may create multiple such tables with different stattab identifiers to hold separate sets of statistics. Additionally, users can maintain different sets of statistics within a single stattab by making use of the statid parameter (which can help avoid cluttering the user's schema).

For all of the set/get procedures, if stattab is not provided (i.e., null), the operation will work directly on the dictionary statistics; therefore, users need not create these statistics tables if they only plan to modify the dictionary directly. However, if stattab is not null, then the set/get operation will work on the specified user statistics table, not the dictionary.

This set of procedures enable the storage and retrieval of individual column-, index-, and table- related statistics.

Procedures in DBMS_STATS

The statistic gathering related procedures in DBMS_STATS are:

PREPARE_COLUMN_VALUES

The procedure prepare_column_vlaues is used to convert user-specified minimum, maximum, and histogram endpoint datatype-specific values into Oracle's internal representation for future storage via set_column_stats.

Generic input arguments:

- srec.epc - The number of values specified in charvals, datevals, numvals, or rawvals. This value must be between 2 and 256 inclusive. Should be set to 2 for procedures which don't allow histogram information (nvarchar and rowid). The first corresponding array entry should hold the minimum value for the column and the last entry should hold the maximum. If there are more than two entries, then all the others hold the remaining height-balanced or frequency histogram endpoint values (with in-between values ordered from next-smallest to next-largest). This value may be adjusted to account for compression, so the returned value should be left as is for a call to set_column_stats.

- srec.bkvals - If a frequency distribution is desired, this array contains the number of occurrences of
- each distinct value specified in charvals, datevals, numvals, or rawvals. Otherwise, it is merely an ouput argument and must be set to null when this procedure is called.

Datatype specific input arguments (one of these):

- charvals - The array of values when the column type is character-based. Up to the first 32 bytes of each string should be provided. Arrays must have between 2 and 256 entries, inclusive.
- datevals - The array of values when the column type is date-based.
- numvals - The array of values when the column type is numeric-based.
- rawvals - The array of values when the column type is raw. Up to the first 32 bytes of each strings should be provided.
- nvmin,nvmax - The minimum and maximum values when the column type is national character set based (NLS). No histogram information can be provided for a column of this type.
- rwmin,rwmax - The minimum and maximum values when the column type is rowid. No histogram information can be provided for a columns of this type.

Output arguments:

- srec.minval - Internal representation of the minimum which is suitable for use in a call to set_column_stats.
- srec.maxval - Internal representation of the maximum which is suitable for use in a call to set_column_stats.
- srec.bkvals - array suitable for use in a call to set_column_stats.
- srec.novals - array suitable for use in a call to set_column_stats.

Exceptions:

- ORA-20001: Invalid or inconsistent input values

SET_COLUMN_STATS

The set_column_stats procedure is used to set column-related information.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table to which this column belongs
- colname - The name of the column
- partname - The name of the table partition in which to store the statistics. If the table is partitioned and partname is null, the statistics will be stored at the global table level.
- stattab - The user statistics table identifier describing where to store the statistics. If stattab is null, the statistics will be stored directly in the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- distcnt - The number of distinct values
- density - The column density. If this value is null and distcnt is not null, density will be derived from distcnt.
- nullcnt - The number of nulls
- srec - StatRec structure filled in by a call to prepare_column_values or get_column_stats.
- avgclen - The average length for the column (in bytes)
- flags - For internal Oracle use (should be left as null)
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid or inconsistent input values

SET_INDEX_STATS

The procedure set_index_stats is used to set index-related information.

Input arguments:

- ownname - The name of the schema

- indname - The name of the index
- partname - The name of the index partition in which to store the statistics. If the index is partitioned and partname is null, the statistics will be stored at the global index level.
- stattab - The user statistics table identifier describing where to store the statistics. If stattab is null, the statistics will be stored directly in the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- numrows - The number of rows in the index (partition)
- numlblks - The number of leaf blocks in the index (partition)
- numdist - The number of distinct keys in the index (partition)
- avglblk - Average integral number of leaf blocks in which each distinct key appears for this index (partition). If not provided, this value will be derived from numlblks and numdist.
- avgdblk - Average integral number of data blocks in the table pointed to by a distinct key for this index (partition). If not provided, this value will be derived from clstfct and numdist.
- clstfct - see clustering_factor column of the user_indexes view for a description.
- indlevel - The height of the index (partition)
- flags - For internal Oracle use (should be left as null)
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid input value

SET_TABLE_STATS

The procedure set_table_stats is used to set table-related information

Input arguments:

- ownname - The name of the schema

- tabname - The name of the table
- partname - The name of the table partition in which to store the statistics. If the table is partitioned and partname is null, the statistics will be stored at the global table level.
- stattab - The user statistics table identifier describing where to store the statistics. If stattab is null, the statistics will be stored directly in the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- numrows - Number of rows in the table (partition)
- numblks - Number of blocks the table (partition) occupies
- avgrlen - Average row length for the table (partition)
- flags - For internal Oracle use (should be left as null)
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid input value

CONVERT_RAW_VALUE

The procedure convert_raw_value is used to convert the internal representation of a minimum or maximum value into a datatype-specific value. The minval and maxval fields of the StatRec structure as filled in by get_column_stats or prepare_column_values are appropriate values for input.

Input argument

- rawval - The raw representation of a column minimum or maximum

Datatype specific output arguments:

- resval - The converted, type-specific value

Exceptions:

- None

GET_COLUMN_STATS

The purpose of the procedure get_column_stats is to get all column-related information for a specified table.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table to which this column belongs
- colname - The name of the column
- partname - The name of the table partition from which to get the statistics. If the table is partitioned and partname is null, the statistics will be retrieved from the global table level.
- stattab - The user statistics table identifier describing from where to retrieve the statistics. If stattab is null, the statistics will be retrieved directly from the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- statown - The schema containing stattab (if different then ownname)

Output arguments:

- distcnt - The number of distinct values
- density - The column density
- nullcnt - The number of nulls
- srec - structure holding internal representation of column minimum, maximum, and histogram values
- avgclen - The average length of the column (in bytes)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges or no statistics have been stored for requested object.

GET_INDEX_STATS

The purpose of the ger_index_stats procedure is to get all index-related information for a specified index.

Input arguments:

- ownname - The name of the schema
- indname - The name of the index
- partname - The name of the index partition for which to get the statistics. If the index is partitioned and partname is null, the statistics will be retrieved for the global index level.
- stattab - The user statistics table identifier describing from where to retrieve the statistics. If stattab is null, the statistics will be retrieved directly from the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- statown - The schema containing stattab (if different then ownname)

Output arguments:

- numrows - The number of rows in the index (partition)
- numlblks - The number of leaf blocks in the index (partition)
- numdist - The number of distinct keys in the index (partition)
- avglblk - Average integral number of leaf blocks in which each distinct key appears for this index (partition).
- avgdblk - Average integral number of data blocks in the table pointed to by a distinct key for this index (partition).
- clstfct - The clustering factor for the index (partition).
- indlevel - The height of the index (partition).

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges or no statistics have been stored for requested object

GET_TABLE_STATS

The purpose of the get_table_stats procedure is to get all table-related information for a specified table.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table to which this column belongs
- partname - The name of the table partition from which to get the statistics. If the table is partitioned and partname
- is null, the statistics will be retrieved from the global table level.
- stattab - The user statistics table identifier describing from where to retrieve the statistics. If stattab is null, the statistics will be retrieved directly from the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- statown - The schema containing stattab (if different then ownname)

Output arguments:

- numrows - Number of rows in the table (partition)
- numblks - Number of blocks the table (partition) occupies
- avgrlen - Average row length for the table (partition)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges or no statistics have been stored for requested object

DELETE_COLUMN_STATS

The purpose of the delete_column_stats procedure is to delete column-related statistics for a specified table.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table to which this column belongs
- colname - The name of the column

- partname - The name of the table partition for which to delete the statistics. If the table is partitioned and partname is null, global column statistics will be deleted.
- stattab - The user statistics table ideentifier describing from where to delete the statistics. If stattab is null, the statistics will be deleted directly from the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- cascade_parts - If the table is partitioned and partname is null, setting this to true will cause the deletion of statistics for this column for all underlying partitions as well.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

DELETE_INDEX_STATS

The purpose of the delete_index_stats procedure is to deletes index-related statistics for the specified index.

Input arguments:

- ownname - The name of the schema
- indname - The name of the index
- partname - The name of the index partition for which to delete the statistics. If the index is partitioned and partname is null, index statistics will be deleted at the global level.
- stattab - The user statistics table identifier describing from where to delete the statistics. If stattab is null, the statistics will be deleted directly from the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- cascade_parts - If the index is partitioned and partname is null, setting this to true will cause the deletion of statistics for this index for all underlying partitions as well.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

DELETE_TABLE_STATS

The purpose of the procedure delete_table_stats is to delete table-related statistics from the specified table.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table to which this column belongs
- colname - The name of the column
- partname - The name of the table partition from which to get the statistics. If the table is partitioned and partname is null, the statistics will be retrieved from the global table level.
- stattab - The user statistics table identifier describing from where to retrieve the statistics. If stattab is null, the statistics will be retrieved directly from the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- cascade_parts - If the table is partitioned and partname is null, setting this to true will cause the deletion of statistics for this table for all underlying partitions as well.
- cascade_columns - Indicates that delete_column_stats should be called for all underlying columns (passing the cascade_parts parameter).
- cascade_indexes - Indicates that delete_index_stats should be called for all underlying indexes (passing the cascade_parts parameter).
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

DELETE_SCHEMA_STATS

The purpose of the delete_schema_stats procedure is to delete statistics for a specified schema.

Input arguments:

- ownname - The name of the schema
- stattab - The user statistics table identifier describing from where to delete the statistics. If stattab is null, the statistics will be deleted directly in the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

DELETE_DATABASE_STATS

The purpose of the delete_database_stats procedure is to delete statistics for an entire database.

Input arguments:

- stattab - The user statistics table identifier describing from where to delete the statistics. If stattab is null, the statistics will be deleted directly in the dictionary.
- statid - The (optional) identifier to associate with these statistics within stattab (Only pertinent if stattab is not NULL).
- statown - The schema containing stattab. If stattab is not null and statown is null, it is assumed that every schema in the database contains a user statistics table with the name stattab.

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

The set of procedures that enable the transference of statistics from the dictionary to a user statistics table (export_*) and from a user statistics table to the dictionary (import_*) are:

CREATE_STAT_TABLE

The purpose of the create_stat_table procedure is to create a table with name 'stattab' in 'ownname' schema which is capable of holding statistics. The columns and types that compose this table are not relevant as it should be accessed solely through the procedures in this package.

Input arguments:

- ownname - The name of the schema
- stattab - The name of the table to create. This value should be passed as the 'stattab' argument to other procedures when the user does not wish to modify the dictionary statistics directly.
- tblspace - The tablespace in which to create the statistics tables. If none is specified, they will be created in the user's default tablespace.

Exceptions:

- ORA-20000: Table already exists or insufficient privileges
- ORA-20001: Tablespace does not exist

DROP_STAT_TABLE

The purpose of the drop_stat_table procedure is to drop a user statistics table for a specified user (schema.)

Input arguments:

- ownname - The name of the schema
- stattab - The user statistics table identifier

Exceptions:

- ORA-20000: Table does not exists or insufficient privileges

EXPORT_COLUMN_STATS

The purpose of the export_column_stats procedure is to retrieve statistics for a particular column and store them in the user statistics table identified by the value of stattab.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table to which this column belongs
- colname - The name of the column
- partname - The name of the table partition. If the table is partitioned and partname is null, global and partition column statistics will be exported.
- stattab - The user statistics table identifier describing where to store the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

EXPORT_ INDEX_STATS

The purpose of the export_index_stats procedure is to retrieve statistics for a particular index and store them in the user statistics table identified by the value of stattab.

Input arguments:

- ownname - The name of the schema
- indname - The name of the index
- partname - The name of the index partition. If the index is partitioned and partname is null, global and partition index statistics will be exported.
- stattab - The user statistics table identifier describing where to store the statistics.

- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

EXPORT_TABLE_STATS

The purpose of the export_table_stats is to retrieve statistics for a particular table and store them in the user statistics table (stattab.) Cascade will result in all index and column stats associated with the specified table being exported as well.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table
- partname - The name of the table partition. If the table is partitioned and partname is null, global and partition table statistics will be exported.
- stattab - The user statistics table identifier describing where to store the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- cascade - If true, column and index statistics for this table will also be exported.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

EXPORT_SCHEMA_STATS

The purpose of the export_shema_stats procedure is to retrieve statistics for all objects in the schema identified by ownname and store them in the user statistics table identified by stattab.

Input arguments:

- ownname - The name of the schema
- stattab - The user statistics table identifier describing where to store the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

EXPORT_DATABASE_STATS

Retrieves statistics for all objects in the database and stores them in the user statistics tables identified by statown.stattab

Input arguments:

- stattab - The user statistics table identifier describing where to store the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab. If statown is null, it is assumed that every schema in the database contains a user statistics table with the name stattab.

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

IMPORT_COLUMN_STATS

Retrieves statistics for a particular column from the user statistics table identified by stattab and stores them in the dictionary

Input arguments:

- ownname - The name of the schema

- tabname - The name of the table to which this column belongs
- colname - The name of the column
- partname - The name of the table partition. If the table is partitioned and partname is null, global and partition column statistics will be imported.
- stattab - The user statistics table identifier describing from where to retrieve the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid or inconsistent values in the user statistics table

IMPORT_INDEX_STATS

The purpose of the import_index_stats procedure is to retrieve statistics for a particular index from the user statistics table identified by stattab and store them in the dictionary of the target database.

Input arguments:

- ownname - The name of the schema
- indname - The name of the index
- partname - The name of the index partition. If the index is partitioned and partname is null, global and partition index statistics will be imported.
- stattab - The user statistics table identifier describing from where to retrieve the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges

- ORA-20001: Invalid or inconsistent values in the user statistics table

IMPORT_TABLE_STATS

The purpose of the import_table_stats procedure is to retrieve statistics for a particular table from the user statistics table identified by stattab and store them in the dictionary. Cascade will result in all index and column statistics associated with the specified table being imported as well.

Input arguments:

- ownname - The name of the schema
- tabname - The name of the table
- partname - The name of the table partition. If the table is partitioned and partname is null, global and partition table statistics will be imported.
- stattab - The user statistics table identifier describing from where to retrieve the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- cascade - If true, column and index statistics for this table will also be imported.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid or inconsistent values in the user statistics table

IMPORT_SCHEMA_STATS

The purpose of the import_schema_stats procedure is to retrieve statistics for all objects in the schema identified by ownname from the user statistics table and store them in the dictionary

Input arguments:

- ownname - The name of the schema
- stattab - The user stat table identifier describing from where to retrieve the statistics.

- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid or inconsistent values in the user statistics table

IMPORT_DATABASE_STATS

The purpose of the import_database_stats procedure is to retrieve statistics for all objects in the database from the user statistics table(s) and store them in the dictionary

Input arguments:

- stattab - The user stat table identifier describing from where to retrieve the statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab. If statown is null, it is assumed that every schema in the database contains a user statistics table with the name stattab.

Exceptions:

- ORA-20000: Object does not exist or insufficient privileges
- ORA-20001: Invalid or inconsistent values in the user statistics table

The next set of procedures enable the gathering of certain classes of optimizer statistics with possible performance improvements over the ANALYZE command.

The procedures are:

GATHER_ INDEX_STATS

The purpose of this procedure is to gather index statistics. It is equivalent to running:

```
ANALYZE INDEX [ownname.]indname [PARTITION partname]
COMPUTE STATISTICS |
ESTIMATE STATISTICS SAMPLE estimate_percent PERCENT
```

It does not execute in parallel.

Input arguments:

- ownname - schema of index to analyze
- indname - name of index
- partname - name of partition
- estimate_percent - Percentage of rows to estimate (NULL means compute). The valid range is [0.000001,100). This value may be increased automatically to achieve better results.
- stattab - The user statistics table identifier describing where to save the current statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Index does not exist or insufficient privileges
- ORA-20001: Bad input value

GATHER_TABLE_STATS

The purpose of the gather_table_stats procedure is to gather table and column (and index) statistics. It attempts to parallelize as much of the work as possible, but there are some restrictions as described in the individual parameters. This operation will not parallelize if the user does not have select privilege on the table being analyzed.

Input arguments:

- ownname - schema of table to analyze
- tabname - name of table
- partname - name of partition

- estimate_percent - Percentage of rows to estimate (NULL means compute.) The valid range is [0.000001,100). This value may be increased automatically to achieve better results.
- block_sample - whether or not to use random block sampling instead of random row sampling. Random block sampling is more efficient, but if the data is not randomly distributed on disk then the sample values may be somewhat correlated. Only pertinent when doing an estimate statistics.
- method_opt - method options of the following format (the phrase 'SIZE 1' is required to ensure gathering statistics in parallel and for use with the phrase hidden):
  - FOR ALL [INDEXED | HIDDEN] COLUMNS [SIZE integer]
  - FOR COLUMNS [SIZE integer] column|attribute [,column|attribute ...]
- Optimizer related table statistics are always gathered.
- degree - degree of parallelism (NULL means use table default value)
- granularity - the granularity of statistics to collect (only pertinent if the table is partitioned)
  - 'DEFAULT' - gather global- and partition-level statistics
  - 'SUBPARTITION' - gather subpartition-level statistics
  - 'PARTITION' - gather partition-level statistics
  - 'GLOBAL' - gather global statistics
  - 'ALL' - gather all (subpartition, partition, and global) statistics
- cascade - gather statistics on the indexes for this table. Index statistics gathering will not be parallelized. Using this option is equivalent to running the gather_index_stats procedure on each of the table's indexes.
- stattab - The user statistics table identifier describing where to save the current statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Table does not exist or insufficient privileges

- ORA-20001: Bad input value

GATHER_SCHEMA_STATS

The purpose of the gather_schema_stats procedure is to gather all schema statistics for the specified schema. It attempts to parallelize as much of the work as possible, but there are some restrictions as described in the individual parameters. This operation will not parallelize if the user does not have select privilege on the objects being analyzed.

Input arguments:

- ownname - schema to analyze (NULL means current schema)
- estimate_percent - Percentage of rows to estimate (NULL means compute.) The valid range is [0.000001,100).
- block_sample - whether or not to use random block sampling instead of random row sampling. Random block sampling is more efficient, but if the data is not randomly distributed on disk then the sample values may be somewhat correlated. Only pertinent when doing an estimate statistics.
- method_opt - method options of the following format (the phrase 'SIZE 1' is required to ensure gathering statistics in parallel and for use with the phrase hidden):
  - FOR ALL [INDEXED | HIDDEN] COLUMNS [SIZE integer]
  - This value will be passed to all of the individual tables.
- degree - degree of parallelism (NULL means use table default value)
- granularity - the granularity of statistics to collect (only pertinent if the table is partitioned)
  - 'DEFAULT' - gather global- and partition-level statistics
  - 'SUBPARTITION' - gather subpartition-level statistics
  - 'PARTITION' - gather partition-level statistics
  - 'GLOBAL' - gather global statistics
  - 'ALL' - gather all (subpartition, partition, and global) statistics
- cascade - gather statistics on the indexes as well. Index statistics gathering will not be parallelized. Using this option is equivalent to running the gather_index_stats procedure on each of the indexes in the schema in addition to gathering table and column statistics.

- stattab - The user statistics table identifier describing where to save the current statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- options - further specification of which objects to gather statistics for:
  - 'GATHER' - gather statistics on all objects in the schema
  - 'GATHER STALE' - gather statistics on stale objects as determined by looking at the *_tab_modifications views. Also, return a list of obects found to be stale.
  - 'GATHER EMPTY' - gather statistics on objects which currently have no statistics. also, return a list of objects found to have no statistics.
  - 'LIST STALE' - return list of stale objects as determined by looking at the *_tab_modifications views
  - 'LIST EMPTY' - return list of objects which currently have no statistics
- objlist - list of objects found to be stale or empty
- statown - The schema containing stattab (if different then ownname)

Exceptions:

- ORA-20000: Schema does not exist or insufficient privileges
- ORA-20001: Bad input value

GATHER_DATABASE_STATS

The purpose of the gather_database_stats procedure is to gather all database statistics. It attempts to parallelize as much of the work as possible, but there are some restrictions as described in the individual parameters. This operation will not parallelize if the user does not have select privilege on the objects being analyzed.

Input arguments:

- estimate_percent - Percentage of rows to estimate (NULL means compute.) The valid range is [0.000001,100).
- block_sample - whether or not to use random block sampling instead of random row sampling. Random block sampling is more efficient, but if the data is not randomly distributed on disk then the sample values may

be somewhat correlated. Only pertinent when doing an estimate statistics.

- method_opt - method options of the following format (the phrase 'SIZE 1' is required to ensure gathering statistics in parallel and for use with the phrase hidden):
  - FOR ALL [INDEXED | HIDDEN] COLUMNS [SIZE integer]
  - This value will be passed to all of the individual tables.
- degree - degree of parallelism (NULL means use table default value)
- granularity - the granularity of statistics to collect (only pertinent if the table is partitioned)
  - 'DEFAULT' - gather global- and partition-level statistics
  - 'SUBPARTITION' - gather subpartition-level statistics
  - 'PARTITION' - gather partition-level statistics
  - 'GLOBAL' - gather global statistics
  - 'ALL' - gather all (subpartition, partition, and global) statistics
- cascade - gather statistics on the indexes as well. Index statistics gathering will not be parallelized. Using this option is equivalent to running the gather_index_stats procedure on each of the indexes in the database in addition to gathering table and column statistics.
- stattab - The user stat table identifier describing where to save the current statistics.
- statid - The (optional) identifier to associate with these statistics within stattab.
- options - further specification of which objects to gather statistics for
  - 'GATHER STALE' - gather statistics on stale objects as determined by looking at the *_tab_modifications views. Also, return a list of obects found to be stale.
  - 'GATHER EMPTY' - gather statistics on objects which currently have no statistics. also, return a list of objects found to have no statistics.
  - 'LIST STALE' - return list of stale objects as determined by looking at the *_tab_modifications views
  - 'LIST EMPTY' - return list of objects which currently have no statistics
- objlist - list of objects found to be stale or empty

- statown - The schema containing stattab. If null, it will assume there is a table named stattab in each relevant schema in the
- database if stattab is specified for saving current statistics.

Exceptions:

- ORA-20000: Insufficient privileges
- ORA-20001: Bad input value

Also provided is the following procedure for generating some statistics for derived objects when you have sufficient statistics on related objects.

GENERATE_STATS

The purpose of this procedure is to generate object statistics from previously collected statistics of related objects. For fully populated schemas, the gather procedures should be used instead when more accurate statistics are desired. The currently supported objects are b-tree and bitmap indexes.

Input arguments:

- ownname - schema of object
- objname - name of object
- organized - the amount of ordering associated between the index and its undelrying table. A heavily organized index would have consecutive index keys referring to consecutive rows on disk for the table(the same block). A heavily disorganized index would have consecutive keys referencing different table blocks on disk. This parameter is only used for b-tree indexes. The number can be in the range of 0-10, with 0 representing a completely organized index and 10 a completely disorganized one.

Exceptions:

- ORA-20000: Unsupported object type of object does not exist
- ORA-20001: Invalid option or invalid statistics

Stabilizing Execution Plans in a Data Warehouse in Oracle8i

In versions of Oracle prior to Oracle8i the only way to stablize an execution plan was to ensure that tables where analyzed frequently and that the relative ratios of rows in the tables involved stayed relatively stable. Neither of these options in pre-Oracle8i for stabilizing execution plans worked 100 percent of the time. In Oracle8i a new feature known as OUTLINEs has been added.

New in Oracle8i is the OUTLINE capability. An outline allows the DBA to tune a SQL statement and then store the optimizer plan for the statement in what is known as an OUTLINE. From that point forward whenever an identical SQL statement to the one in the OUTLINE is used, it will use the optimizer instructions contained in the OUTLINE.

This storing of plan outlines for SQL statements is known as plan stability and insures that changes in the Oracle environment don't affect the way a SQL statement is optimized by the cost based optimizer. If you wish, Oracle will define plans for all issued SQL statements at the time they are executed and this stored plan will be reused until altered or dropped. Generally I do not suggest using the automatic outline feature as it can lead to poor plans being reused by the optimizer. It makes more sense to monitor for high cost statements and tune them as required, storing an outline for them only once they have been properly tuned.

As with the storage of SQL in the shared pool, storage of outlines depends on the statement being reissued in an identical fashion each time it is used. If even one space is out of place the stored outline is not reused. Therefore your queries should be stored as PL/SQL procedures, functions or packages (or perhaps Java routines) and bind variables should always be used. This allows reuse of the stored image of the SQL as well as reuse of stored outlines.

Remember that to be useful over the life of an application the outlines will have to be periodically verified by checking SQL statement performance. If performance of SQL statements degrades the stored outline may have to be dropped and regenerated after the SQL is retuned.

Creation of a OUTLINE object

Outlines are created using the CREATE OUTLINE command, the syntax for this command is:

```
CREATE [OR REPLACE] OUTLINE outline_name
[FOR CATEGORY category_name]
ON statement;
```

Where:

- Outline_name -- is a unique name for the outline
- [FOR CATEGORY category_name] – This optional clause allows more than one outline to be associated with a single query by specifying multiple catagories each named uniquely.
- ON statement – This specifies the statement for which the outline is prepared.

An example would be:

```
CREATE OR REPLACE OUTLINE get_tables
ON
SELECT
a.owner,
a.table_name,
a.tablespace_name,
SUM(b.bytes),
COUNT(b.table_name) extents
FROM
       dba_tables a,
       dba_extents b
WHERE
       a.owner=b.owner
       AND a.table_name=b.table_name
GROUP BY
       a.owner, a.table_name, a.tablespace_name;
```

Assuming the above select is a part of a stored PL/SQL procedure or perhaps part of a view, the stored outline will now be used each time an exactly matching SQL statement is issued.

Altering a OUTLINE

Outlines are altered using the ALTER OUTLINE or CREATE OR REPLACE form of the CREATE command. The format of the command is identical whether it is used for initial creation or replacement of an existing outline. For example, what if we want to add SUM(b.blocks) to the previous example?

```
CREATE OR REPLACE OUTLINE get_tables
ON
SELECT
a.owner,
a.table_name,
a.tablespace_name,
SUM(b.bytes),
COUNT(b.table_name) extents,
SUM(b.blocks)
FROM
       dba_tables a,
       dba_extents b
```

```
WHERE
        a.owner=b.owner
        AND a.table_name=b.table_name
GROUP BY
        a.owner, a.table_name, a.tablespace_name;
```

The above example has the effect of altering the stored outline *get_tables* to include any changes brought about by inclusion of the SUM(b.blocks) in the SELECT list. But what if we want to rename the outline or change a category name? The ALTER OUTLINE command has the format:

```
ALTER OUTLINE outline_name
[REBUILD]
[RENAME TO new_outline_name]
[CHANGE CATEGORY TO new_category_name]
```

The ALTER OUTLINE command allows us to rebuild the outline for an existing outline_name as well as rename the outline or change its category. The benefit of using the ALTER OUTLINE command is that we do not have to respecify the complete SQL statement as we would have to using the CREATE OR REPLACE command.

Dropping an OUTLINE

Outlines are dropped using the DROP OUTLINE command the syntax for this command is:

```
DROP OUTLINE outline_name;
```

Use of the OUTLN_PKG To Manage SQL Stored Outlines

The OUTLN_PKG package provides for the management of stored outlines. A stored outline is an execution plan for a specific SQL statement. A stored outline permits the optimizer to stabilize a SQL statements execution plan giving repeatable execution plans even when data and statistics change.

The DBA should take care to who they grant execute on the OUTLN_PKG, by default it is not granted to the public user group nor is a public synonym created.

The following sections show the packages in the OUTLN_PKG.

DROP_UNUSED

The drop_unused procedure is used to drop outlines that have not been used in the compilation of SQL statements. The drop_unused procedure has no arguments.

```
SQL> EXECUTE OUTLN_PKG.DROP_UNUSED;

PL/SQL procedure successfully executed.
```

To determine if a SQL statement OUTLINE is unused, perform a select against the DBA_OUTLINES view:

```
SQL> desc dba_outlines;
 Name                           Null?    Type
 ------------------------------ -------- ----
 NAME                                    VARCHAR2(30)
 OWNER                                   VARCHAR2(30)
 CATEGORY                                VARCHAR2(30)
 USED                                    VARCHAR2(9)
 TIMESTAMP                               DATE
 VERSION                                 VARCHAR2(64)
 SQL_TEXT                                LONG

SQL> set long 1000
SQL> select * from dba_outlines where used='UNUSED';

NAME         OWNER  CATEGORY USED   TIMESTAMP VERSION    SQL_TEXT
------------ ------ -------- ------ --------- ---------- ----------------------
TEST_OUTLINE SYSTEM TEST     UNUSED 08-MAY-99 8.1.3.0.0  select a.table_name,
                                                         b.tablespace_name,
                                                         c.file_name from
                                                         dba_tables a,
                                                         dba_tablespaces b,
                                                         dba_data_files c
                                                         where
                                                         a.tablespace_name =
                                                         b.tablespace_name
                                                         and b.tablespace_name
                                                         = c.tablespace_name
                                                         and c.file_id =
                                                         (select
                                                         min(d.file_id) from
                                                         dba_data_files d
                                                         where
                                                         c.tablespace_name =
                                                         d.tablespace_name)


1 row selected.

SQL> execute sys.outln_pkg.drop_unused;

PL/SQL procedure successfully completed.

SQL> select * from dba_outlines where used='UNUSED';

no rows selected
```

Remember, the procedure drops all unused outlines so use it carefully.

DROP_BY_CAT

The drop_by_cat procedure drops all outlines that belong to a specific category. The procedure drop_by_cat has one input variable, cat, a VARCHAR 2 that corresponds to the name of the category you want to drop.

```
SQL> create outline test_outline for category test on
  2  select a.table_name, b.tablespace_name, c.file_name from
  3  dba_tables a, dba_tablespaces b, dba_data_files c
  4  where
  5  a.tablespace_name=b.tablespace_name
  6  and b.tablespace_name=c.tablespace_name
  7  and c.file_id = (select min(d.file_id) from dba_data_files d
  8  where c.tablespace_name=d.tablespace_name)
  9  ;
Operation 180 succeeded.

SQL> select * from dba_outlines where category='TEST';
NAME         OWNER  CATEGORY USED   TIMESTAMP VERSION    SQL_TEXT
------------ ------ -------- ------ --------- ---------- -------------------------
TEST_OUTLINE SYSTEM TEST     UNUSED 08-MAY-99 8.1.3.0.0  select a.table_name, b.ta
                                                         blespace_name, c.file_nam
                                                         e from
                                                         dba_tables a, dba_tablesp
                                                         aces b, dba_data_files c
                                                         where
                                                         a.tablespace_name=b.table
                                                         space_name
                                                         and b.tablespace_name=c.t
                                                         ablespace_name
                                                         and c.file_id = (select m
                                                         in(d.file_id) from dba_da
                                                         ta_files d
                                                         where c.tablespace_name=d
                                                         .tablespace_name)



1 row selected.

SQL> execute sys.outln_pkg.drop_by_cat('TEST');

PL/SQL procedure successfully completed.

SQL> select * from dba_outlines where category='TEST';

no rows selected
```

UPDATE_BY_CAT

The update_by_cat procedure changes all of the outlines in one category to a new category. If the SQL text in an outline already has an outline in the target category, then it is not merged into the new category. The procedure has two input variables, oldcat VARCHAR2 and newcat VARCHAR2 where oldcat corresponds to the category to be merged and newcat is the new category that oldcat is to be merged with.

```
SQL> create outline test_outline for category test on
  2  select a.table_name, b.tablespace_name, c.file_name from
  3  dba_tables a, dba_tablespaces b, dba_data_files c
  4  where
  5  a.tablespace_name=b.tablespace_name
  6  and b.tablespace_name=c.tablespace_name
  7  and c.file_id = (select min(d.file_id) from dba_data_files d
  8  where c.tablespace_name=d.tablespace_name)
  9  ;

Operation 180 succeeded.

SQL> create outline test_outline2 for category test on
  2  select * from dba_data_files;

Operation 180 succeeded.

SQL> create outline prod_outline1 for category prod on
  2  select owner,table_name from dba_tables;

Operation 180 succeeded.

SQL> create outline prod_outline2 for category prod on
  2  select * from dba_data_files;

Operation 180 succeeded.

SQL>  select name,category from dba_outlines order by category
NAME           CATEGORY
-------------- --------
PROD_OUTLINE1  PROD
PROD_OUTLINE2  PROD
TEST_OUTLINE2  TEST
TEST_OUTLINE   TEST

4 rows selected.

SQL> execute sys.outln_pkg.update_by_cat('TEST','PROD');

PL/SQL procedure successfully completed.

SQL> select name,category from dba_outlines order by category;
NAME           CATEGORY
-------------- --------
TEST_OUTLINE   PROD
PROD_OUTLINE1  PROD
PROD_OUTLINE2  PROD
TEST_OUTLINE2  TEST

4 rows selected.
```

As a result of the update_by_cat procedure call we moved the TEST_OUTLINE outline into the PROD category, but the TEST_OUTLINE2, since it is a duplicate of PROD_OUTLINE2, was not merged.

Oracle8i Materialized Views, Summaries and Data Warehousing

The concept of snapshots, not particularily useful in data warehousing (unless you snapshot to a datamart) has been expanded in Oracle8i to include materialized views. Materialized views are similar to snapshots except that they can reside in the same database as their master table(s). Another powerful feature of materialized views is that they can be the subject of DIMENSIONS, a new concept in Oracle8i that explains to the optimizer how a hierarchical or parent-child relationship in a materialized view is constructed, thus allowing query re-write. Query re-write is when the optimizer recognizes that a query on a materialized views base table(s) can be re-written to use the materialized view to operate more efficiently.

As with snapshots, a materialized view can have a materialized view log to speed refresh operations. Since the materialized view log must exist first before a materialized view will use it, let's look at materialized view logs first.

MATERIALIZED VIEW LOGS IN Oracle8i

In order to facilitate fast refreshes of materialized views, you must create a materialized view log. If a materialized view log is not available for a materialized view, a fast refresh cannot be done and a complete refresh is the only refresh mechanism. A complete refresh involves truncating the materialized view table and then repopulating the materialized view by re-executing its build query. A fast refresh uses the information in the materialized view log to only apply changes to the materialized view.

Creation of a Materialized View Log

The actual syntax of the CREATE MATERIALIZED VIEW LOG command is rather lengthy so I will refer you to the SQL manual for the details. I will however show an example of the creation of a materialized view log.

```
CREATE MATERIALIZED VIEW LOG ON tab_example1
STORAGE (INITIAL 250K NEXT 250K PCTINCREASE 0)
TABLESPACE view_logs
PARALLEL 4
NOLOGGING
NOCACHE
WITH PRIMARY KEY, ROWID (tablespace_name)
INCLUDING NEW VALUES;
```

Don’t be confused when it tells you that a snapshot log has been created, remember that snapshots and materialized views are synonymous. The above command creates a materialized view log (or snapshot log if you prefer) on the table tab_example1 using the specified storage parameters and tablespace. The

log will use a parallel degree of 4 (normally this is not required since the PARALLEL clause by itself will cause Oracle to calculate the proper degree.) The log is not logged, the default is LOGGED (which means it is not recoverable) and will not be cached, the default is CACHED, in the SGA. The log will track both primary key and rowid information as well as any changes to the filter column TABLESPACE_NAME. Filter columns cannot be part of the primary key and must exist in the master table. If no WITH clause is specified, the primary key values are tracked by default. Materialized view logs can be partitioned just like regular tables.

Altering A Materialized View Log

All aspects of a materialized view log are alterable, again I will refer you to the CD-ROM for the details. This includes the adding of filter columns, rowid or primary key data to that which is all ready being stored in the log.

Dropping a Materialized View Log

The command for dropping a materialized view log is simple:

```
DROP MATERIALIZED VIEW LOG ON [schema.]tablename;
```

MATERIALZED VIEWS (Snapshots) IN Oracle8i

Another feature of Oracle that needs administration is the snapshot (also known as a materialized view.) Snapshots are copies of either an entire single table or set of its rows (simple snapshot) or a collection of tables, views, or their rows using joins, grouping, and selection criteria (complex snapshots). Snapshots are very useful in a distributed environment where remote locations need a queriable copy of a table from the master database. Instead of paying the penalty for using the network to send out the query and get back the data, the query is against a local table image and is thus much faster. With later versions of Oracle7 and in Oracle8 and Oracle8i, snapshots can be made updatable. As was stated above, the new materialized view is actually a special form of “same database” snapshot.

Snapshots and materialized views are asynchronous in nature; they reflect a table’s or a collection’s state at the time the snapshot was taken. A simple snapshot or materialized view can be periodically refreshed by either use of a snapshot log containing only the changed rows for the snapshot (fast refresh), or a totally new copy (complete refresh). In most cases, the fast refresh is quicker and just as accurate. A fast refresh can only be used if the snapshot or materialized view has a log, and that log was created prior to the creation or last refresh of the snapshot. For a complex snapshot or materialized view, a

complete refresh is required. It is also possible to allow the system to decide which to use, either a fast or complete refresh.

One problem with a snapshot or materialized view log is that it keeps a copy of each and every change to a row. Therefore, if a row undergoes 200 changes between one refresh and the next, there will be 200 entries in the snapshot or materialized view log that will be applied to the snapshot at refresh. This could lead to the refresh of the snapshot or materialized view taking longer than a complete refresh. Each snapshot or materialized view should be examined for the amount of activity it is seeing and if this is occurring with any of them, the snapshot or materialized view log should be eliminated or the refresh mode changed to COMPLETE.

A materialized view is simply a snapshot that is contained in the current instance instead of a remote instance. Other than the keyword MATERIALIZED VIEW the CREATE SNAPSHOT and CREATE SNAPSHOT LOG commands are identical to the CREATE MATERIALIZED VIEW and CREATE MATERIALIZED VIEW LOG commands. Since the CREATE MATERIALIZED VIEW command creates a view, table and an index to maintain the materialized view you must have the CREATE VIEW, CREATE TABLE, CREATE INDEX and CREATE MATERIALIZED VIEW or CREATE SNAPSHOT privileges to create a materialized view. If you wish query rewrite to be available on the materialized views created, the owner of the underlying tables and the materialized view must have QUERY REWRITE or, the creator of the materialized view must have GLOBAL QUERY REWRITE privilege.

In a data warehousing situation a materialized view can be used by Oracle to re-write queries on the fly that the optimizer determines would profit from using the materialized view rather than the base tables. You should take this into consideration when the concurrency of the data is important since a materialized view is only as current as its last refresh.

Creating a Materialized View or Snapshot

The format for the CREATE MATERIALIZED VIEW command is complex enough that I will refer you once again to the CD-ROM SQL Manual. However we will look at a couple of examples. In perusing the manuals in one location it states that the first step in creating a materialized view is to create a DIMENSION. However, in investigating this claim I found no way to tie a DIMENSION to a MATERIALIZED VIEW and that DIMENSIONS are really only of use in data warehousing were rollup and aggregation are of importance. I will touch on DIMENSION creation in a later section of this chapter, however, note that there are no direct ties between MATERIALIZED VIEWs and DIMENSIONs. Perhaps the Oracle8i database engine itself ties them together, but one is not required for the other to function as far as I can determine. Let's get on with some (albeit simple) examples.

In the first example lets do some summary work against the DBA_ views so we can query about total space, total extents, etc. without having to place the code into our reports. This will actually be a materialized view based on two example tables TAB_EXAMPLE1 and TAB_EXAMPLE1 that are based on the underlying DBA_ views DBA_TABLES and DBA_EXTENTS.

```
CREATE MATERIALIZED VIEW table_extents
TABLESPACE tools
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0)
NOLOGGING
BUILD IMMEDIATE
REFRESH COMPLETE START WITH SYSDATE NEXT SYSDATE+1/24
AS
SELECT
  a.owner             owner,
  a.table_name        table_name,
  a.tablespace_name   tablespace_name,
  count(b.owner)      extents,
  sum(b.bytes)        bytes
FROM
tab_example1 a,
tab_example2 b
WHERE
a.owner <>'SYSTEM'
AND a.owner=b.owner
AND a.table_name=b.segment_name
AND b.segment_type='TABLE'
GROUP BY
a.owner,a.table_name, a.tablespace_name;
```

What does a materialized view buy us as far as performance? Let's look at the explain plans for a query on a regular view and then one on the materialized view we just created. First create an identical normal view:

```
CREATE VIEW test_view
AS
SELECT
a.owner owner,
a.table_name table_name,
a.tablespace_name tablespace_name,
count(b.owner) extents,
sum(b.bytes) bytes
FROM
tab_example1 a, tab_example2 b
WHERE
a.owner <>'SYSTEM'
AND a.owner=b.owner
AND a.table_name=b.segment_name
AND b.segment_type='TABLE'
GROUP BY a.owner,a.table_name, a.tablespace_name;
```

Now let's set autotrace on with the explain option and see what happens when we select against each of these objects.

```
SQL> set autotrace on explain
SQL> select * from test_view
2*  where extents>1

OWNER  TABLE_NAME           TABLESPACE_NAME    EXTENTS      BYTES
------ -------------------- --------------- ---------- ----------
SYS    ACCESS$              SYSTEM                   8     536576
SYS    ARGUMENT$            SYSTEM                  10    1191936
SYS    COM$                 SYSTEM                   7     368640
SYS    CON$                 SYSTEM                   3      45056
SYS    DEPENDENCY$          SYSTEM                   7     352256
SYS    ERROR$               SYSTEM                   2      24576
SYS    EXTENT_TO_OBJECT_TAB SYSTEM                   3      45056
SYS    EXT_TO_OBJ           SYSTEM                   4      86016
SYS    HIST_HEAD$           SYSTEM                   3      45056
SYS    IDL_CHAR$            SYSTEM                   7     368640
SYS    IDL_SB4$             SYSTEM                   9     802816
SYS    IDL_UB1$             SYSTEM                  14    5861376
SYS    IDL_UB2$             SYSTEM                  13    3915776
SYS    OBJ$                 SYSTEM                   7     352256
SYS    OBJAUTH$             SYSTEM                   3      45056
SYS    PROCEDURE$           SYSTEM                   2      24576
SYS    SEQ$                 SYSTEM                   2      24576
SYS    SOURCE$              SYSTEM                  18   29503488
SYS    SYN$                 SYSTEM                   3      45056
SYS    VIEW$                SYSTEM                  10    1191936

20 rows selected.

Execution Plan
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   VIEW OF 'TEST_VIEW'
   2    1     FILTER
   3    2       SORT (GROUP BY)
   4    3         MERGE JOIN
   5    4           SORT (JOIN)
   6    5             TABLE ACCESS (FULL) OF 'TAB_EXAMPLE2'
   7    4           SORT (JOIN)
   8    7             TABLE ACCESS (FULL) OF 'TAB_EXAMPLE1'


SQL> select * from table_extents
2* where extents>1
```

```
OWNER  TABLE_NAME           TABLESPACE_NAME    EXTENTS      BYTES
------ -------------------- --------------- ---------- ----------
SYS    ACCESS$              SYSTEM                   8     536576
SYS    ARGUMENT$            SYSTEM                  10    1191936
SYS    COM$                 SYSTEM                   7     368640
SYS    CON$                 SYSTEM                   3      45056
SYS    DEPENDENCY$          SYSTEM                   7     352256
SYS    ERROR$               SYSTEM                   2      24576
SYS    EXTENT_TO_OBJECT_TAB SYSTEM                   3      45056
SYS    EXT_TO_OBJ           SYSTEM                   4      86016
SYS    HIST_HEAD$           SYSTEM                   3      45056
SYS    IDL_CHAR$            SYSTEM                   7     368640
SYS    IDL_SB4$             SYSTEM                   9     802816
SYS    IDL_UB1$             SYSTEM                  14    5861376
SYS    IDL_UB2$             SYSTEM                  13    3915776
SYS    OBJ$                 SYSTEM                   7     352256
SYS    OBJAUTH$             SYSTEM                   3      45056
SYS    PROCEDURE$           SYSTEM                   2      24576
SYS    SEQ$                 SYSTEM                   2      24576
SYS    SOURCE$              SYSTEM                  18   29503488
SYS    SYN$                 SYSTEM                   3      45056
SYS    VIEW$                SYSTEM                  10    1191936

20 rows selected.

Execution Plan
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'MVIEW_TEST'
```

As you can see, we get identical results but the second query against the materialized view only does a single table scan against the materialized view table, not two table scans against the under lying base tables. The results would be even more advantageous for a remote snapshot since no network traffic would be involved. Also notice in the materialized view we are updating once an hour. While a view will give an instantaneous result (after the view itself is instantiated) the materialized view will only be as current as its last refresh. The materialized view can be created such that any commit against the base table forces a refresh against the materialized view if the materialized view is not an has no aggregations or includes no joins.

Altering a Materialized View or Snapshot

As with snapshots, a materialized view can have its physical attributes altered, index parameters changed, its logging and cache parameters changed (look at the syntax for the command on the included CD-ROM SQL Manual) in addition, a materialized view can have the ability to allow query re-write enabled or disabled.

Dropping a Materialized View

The command to drop a materialized view or snapshot is rather simple:

```
DROP MATERIALIZED VIEW|SNAPSHOT
[schema.]materialized_view_name|snapshot_name;
```

Refreshing Materialized Views

Normally a materialized view will be refreshed using the DBMS_JOB queues. This means that you must have at least one job queue set up and operating, normally I suggest at least two queues or mre be set up using the JOB_QUEUE_PROCESSES and JOB_QUEUE_INTERVAL initialization parameters. These parameters are synonymous with the SNAPSHOT_QUEUE_PROCESSES and SNAPSHOT_INTERVAL parameters in prior releases. A third parameter, JOB_QUEUE_KEEP_CONNETIONS forces the database links opened for remote snapshots or materialized views to be held open between refreshes.

Materialized views can be refreshed using COMPLETE, FAST, FORCE, ON DEMAND or ON COMMIT depending on the complexity of the materialized view. A COMPLETE truncates the materialized view table and reloads it from scratch. A FAST uses a materialized view log to only update changed rows. If you intend to use the FAST refresh method, you must create the materialized view log first and then the materialized view. A FORCE will perform a FAST if possible and a COMPLETE if required. ON DEMAND uses the DBMS_MVIEW or DBMS_SNAP packages to complete a refresh and ON COMMIT refreshes a materialized view or snapshot whenever a commit is executed against the base table (for a simple materialized view with no joins or aggregations.)

Oracle8i has provided a new package, DBMS_MVIEW which handles refresh activity on materialized views on demand.

The DBMS_SUMMARY Package in Oracle8i

In order to make the use of DIMENSION and Materialized view objects easier in Oracle8i, Oracle Corporation has provided the DBMS_SUMMARY package. However, the package is synonymed to be called DBMS_OLAP when you look into the data dictionary to find it.

The DBMS_OLAP package is used to maintain summaries. Summaries are also known as materialized views. The package DBMS_OLAP contains functions and procedures used to analyze the effectiveness of materialized views and provide advice as to how to better utilize materialized views. You should be able to call the procedures and functions inside DBMS_OLAP from other packages.

The DBMS_OLAP package can generate the following errors:

Error	Description
ORA-30476	PLAN_TABLE does not exist in the user's schema
ORA-30477	The input select_clause is incorrectly specified
ORA-30478	Specified dimension does not exist
ORA-30479	Summary Advisor error\n <QSM message with more details
QSM-00501	Unable to initialize Summary Advisor environment
QSM-00502	OCI error
QSM-00503	Out of memory
QSM-00504	Internal error
QSM-00505	Syntax error in <parse_entity_name> - <error_description>
QSM-00506	No fact-tables could be found
QSM-00507	No dimensions could be found
QSM-00508	Statistics missing on tables/columns
QSM-00509	Invalid parameter - <parameter_name>
QSM-00510	Statistics missing on summaries
QSM-00511	Invalid fact-tables specified in fact-filter
QSM-00512	Invalid summaries specified in the retention-list
QSM-00513	One or more of the workload tables is missing

There are numerous procedures and functions that make creation and utilization of materialized views easier in the DBMS_OLAP package. The actual package is called DBMS_SUMMARY and the DBMS_OLAP synonym is used to point to DBMS_SUMMARY. The package is created with the DBMSSUM.SQL script.

CLEANUP_SUMDELTA

The procedure cleanup_delta is an internal use procedure and won't be used by the DBA.

COMPUTE_AVG

The function COMPUTE_AVG accepts four input variables: fullsum_avg (NUMBER), fullsum_count(NUMBER), deltasum_avg(NUMBER) and deltasum_count(NUMBER) and returns a NUMBER value. In this form the calculation would seem to have little use, you provide it the average value and the number of measurements along with an average difference from the average and the number of points different and it calculates an adjusted average for example:

```
SQL> select dbms_olap.compute_avg(20,5,1,1) from dual;
DBMS_OLAP.COMPUTE_AVG(20,5,1,1)
-------------------------------
                     16.8333333

SQL> select dbms_olap.compute_avg(20,5,0,0) from dual;
DBMS_OLAP.COMPUTE_AVG(20,5,0,0)
-------------------------------
                             20

SQL> select dbms_olap.compute_avg(25,5,5,5) from dual
DBMS_OLAP.COMPUTE_AVG(25,5,5,5)
-------------------------------
                             15

SQL> select dbms_olap.compute_avg(25,5,5,1) from dual
DBMS_OLAP.COMPUTE_AVG(25,5,5,1)
-------------------------------
                     21.6666667
```

This appears to be an internal use function but those statisticians out there may find it useful.

COMPUTE_AVG2

The function compute_avg2 accepts four input variables: fullsum_sum (number), fullsum_count (number), deltasum_sum (number), deltasum_count (number) and returns a number value. This appears to be a more standard average calculation in that you give it a sum and a count and it returns an average value. If you also specify a sum of differences and the count of values differing from average it produces an adjusted average. For example:

```
SQL> select dbms_olap.compute_avg2(25,5,5,1) from dual
DBMS_OLAP.COMPUTE_AVG2(25,5,5,1)
--------------------------------
                               5

SQL> select dbms_olap.compute_avg2(25,5,0,0) from dual
DBMS_OLAP.COMPUTE_AVG2(25,5,0,0)
--------------------------------
                               5

SQL> select dbms_olap.compute_avg2(25,5,20,1) from dual
DBMS_OLAP.COMPUTE_AVG2(25,5,20,1)
---------------------------------
                             7.5

SQL> select dbms_olap.compute_avg2(25,5,10,5) from dual
DBMS_OLAP.COMPUTE_AVG2(25,5,10,5)
---------------------------------
                             3.5
```

Again this isn't documented in the standard Oracle documentation so it qualifies as an internal use procedure so use it with care.

COMPUTE_STDDEV

The compute_stddev function accepts six input variables: fullsum_stddev (number), fullsum_sum (number),fullsum_count (number), deltasum_stddev (number), deltasum_sum (number) and deltasum_count (number) and returns a number. Again, if you have to provide this function the standard deviations (fullsum_stddev and deltasum_stddev) what exactly is it calculating? This is another undocumented internal use function.

COMPUTE_VARIANCE

The compute_variance function accepts six input variables: fullsum_variance (number), fullsum_sum (number),fullsum_count (number), deltasum_variance (number), deltasum_sum (number) and deltasum_count (number) and returns a number. Again, if you have to provide this function the variances (fullsum_variance and deltasum_variance) what exactly is it calculating? This is another undocumented internal use function.

DISABLE_DEPENDENT and ENABLE_DEPENDENT

These procedures accept a VARCHAR2 comma separated list (detail_tables) of dependent tables and disables the or enables the dependencies. This is another undocumented internal use procedure set.

ESTIMATE_SUMMARY_SIZE

The estimate_summary_size procedure requires that a plan_table be present and that you have select privileges on all underlying objects. None of the underlying objects can be a view. The estimate_summary_size procedure accepts two input variables: stmt_id (varchar2) and select_clause (varchar2) the procedure generates two output variables: num_rows (number) and num_bytes (number). An example PL/SQL anonymous block demonstrating the use of estimate summary size is shown in the next example.

```
SET LONG 1000
DECLARE
stmt_id VARCHAR2(60);
select_cls VARCHAR2(1000);
num_rows NUMBER;
num_bytes NUMBER;
BEGIN
  stmt_id:='object_View';
  select_cls:='SELECT a.owner,a.object_name,
```

```
DECODE(b.tablespace_name,null,c.tablespace_name,b.tablespace_name)
tablespace,
                      b.num_rows
                      FROM test_size a, test_size2 b, test_size3 c
                     WHERE a.owner=b.owner
                     AND a.owner=c.owner
                     AND (a.object_name=b.table_name OR
a.object_name=c.index_name)';
  dbms_olap.estimate_summary_size(stmt_id,select_cls,num_rows,num_bytes);
  dbms_output.put_line(stmt_id||':'||
    ' rows:'||to_char(num_rows)||
    ' bytes:'||to_char(num_bytes));
END;
/

SQL> @test_size
object_View: rows:3293 bytes:691530

PL/SQL procedure successfully completed.
```

The test_size, test_size2 and test_size3 tables are just SELECT * from DBA_OBJECTS, DBA_TABLES and DBA_INDEXES. A test building both a table and a materialized view using the above select with a storage clause of (INITIAL 100K NEXT 100k PCTINCREASE 0) and the default SYSTEM tablespace options for PCTFREE, PCTUSED INITRANS and MAXTRANS resulted in a table sized at 7,397,376 bytes, and a materialzed view at a size of 7,385,088 bytes. Obviously this calculation is just a bit off (only a factor of ten or so…) so be careful relying on it.

EVALUATE_UTILIZATION and EVALUATE_UTILIZATION_W

The procedures evaluate_utilization and evaluate_utilization_w both populate the table MVIEW$_EVALUATIONS table. The table is truncated if it is already populated. Both require RPC connections to an external agent although why since materialized views are usually internalized to the local database I am not sure. Neither of the procedures have input arguments nor do they return a value. The evaluate_utilization_w evaluates utilization based on values entered into the workload profile views WORK$_IDEAL_MVIEW and WORK$_MVIEW_USAGE. The workload profiles are created by the Oracle trace facility.

RECOMMEND_MV and RECOMMEND_MV_W

These procedures generate a set of recommendations about which materialized views should be created, retained, or dropped, based on an analysis of table and column cardinality statistics gathered by ANALYZE.

The recommendations are based on a hypothetical workload in which all possible queries in the data warehouse are weighted equally. This procedure does not require or use the workload statistics tables collected by Oracle Trace, but it works even if those tables are present.

Dimensions must have been created, and there must be foreign key constraints that link the dimensions to fact tables.

Recommending materialized views with a hypothetical workload is appropriate in a DBA-less environment where ease of use is the primary consideration; however, if a workload is available in the default schema, it should be used.

The recommend_mv procedure has the input variables:

- fact_table_filter (varchar2),
- storage_in_bytes (number),
- retention_list (varchar2),
- retention_pct (number with a default of 50)

The procedure populates the MVIEW$_RECOMMENDATION table. The input variables are define as:

- fact_table_filter -- Comma-separated list of fact table names to analyze, or NULL to analyze all fact tables.
- storage_in_bytes -- Maximum storage, in bytes, that can be used for storing materialized views. This number must be non-negative.
- retention_list -- Comma-separated list of materialized view table names. A drop recommendation is not made for any materialized view that appears in this list.
- retention_pct -- Number between 0 and 100 that specifies the percent of existing materialized view storage that must be retained, based on utilization on the actual or hypothetical workload.

A materialized view is retained if the cumulative space, ranked by utilization, is within the retention threshold specified (or if it is explicitly listed in retention_list). Materialized views that have a NULL utilization (e.g.,non-dimensional materialized views) are always retained.

The ouput is gathered into the MVIEW$_RECOMMENDATION view which returns the recommendations made, including a size estimate and the SQL

required to build the materialized view. This is implicit, because it is supplied to the procedure when the procedure is called.

RECOMMEND_MV_W procedure

The recommend_mv_w procedure generates a set of recommendations about which materialized views should be created, retained, or dropped, based on information stored in the workload tables (gathered by Oracle Trace), and an analysis of table and column cardinality statistics gathered by ANALYZE.
RECOMMEND_MV_W requires that you have run ANALYZE to gather table and column cardinality statistics, have collected and formatted the workload statistics, and have created dimensions. The workload is aggregated to determine the count of each request in the workload, and this count is used as a weighting factor during the optimization process.

The domain of all dimensional materialized views that include the specified fact tables identifies the set of materialized views that optimize performance across the workload.

This procedure also creates the WORK$_IDEAL_MVIEW and WORK$_MVIEW_USAGE views.

The procedure has the following input values:

- fact_table_filter (varchar2)
- storage_in_bytes (number)
- retention_list (varchar2)
- retention_pct (number with a default of 80)

The ouput is placed in the MVIEW$_RECOMMENDATIONS table and into the WORK$_IDELA_MVIEW and WORK$_MVIEW_USAGE views. The input parameters are the same as for the RECOMMEND_MV procedure.

The procedures outputs are:

- MVIEW$_RECOMMENDATION -- an OUT variable -- Returns the recommendations made, including a size estimate and the SQL required to build the materialized view. This is implicit, because it is supplied to the procedure when the procedure is called.
- V_192216243_F_5_E_14_8_1(required) -- an IN variable -- Table of workload requests logged by Oracle Trace. This is implicit, because it is supplied to the procedure when the procedure is called.

- V_192216243_F_5_E_15_8_1(required) – an IN variable -- Table of materialized view usages logged by Oracle Trace. This is implicit, because it is supplied to the procedure when the procedure is called.

VERIFY_DIMENSION

The verify_dimension procedure is used to verify the relationships in a dimension are correct. The procedure has four input variables: dimension_name (varchar2), dimension_owner (varchar2), incremental (boolean with a default of TRUE) and check_nulls (boolean with a default of FALSE). The parameters have the following definitions:

- Dimension_name – This is the name of the dimension to verify
- Dimension_owner – This is the name of the dimension owner (schema)
- Incremental – This tells Oracle to perform the tests only for the rows specified in the sumdelta$ table for tables of this dimension; if FALSE check all rows.
- Check_nulls – This tells Oracle if TRUE to check all level columns are non-null; if FALSE this check is omitted. Specify FALSE when all not null columns are enforced with not null constraints.

The procedure verify_dimension has one exception, DimensionNotFound which is raised when the specified dimension is not available.

DIMENSION Objects in Oracle8i

DIMENSION objects are used in data warehouse, DSS and Datamart type applications to provide information on how tables that are denormalized relate to themselves. The CREATE DIMENSION command specifies level and hierarchy information for a table or set of related tables. If you want to use query rewrite with the Oracle optimizer and materialized views you must specify dimensions that the optimizer then uses to "decrypt" the inter and intra-table levels and hierarchies. As an administrator you should know how these DIMENSION objects are created, altered and dropped and we will discuss these topics and show some simple examples. Much beyond the basics I suggest reviewing the application development manuals and the cartridge developer manuals.

Creation of DIMENSION Objects

The CREATE DIMENSION command is used to create dimensions in Oracle8i. The CREATE DIMENSION clause has the following syntax:

```
CREATE [FORCE|NOFORCE] DIMENSION [schema.]dimension_name
Level_clauses
[Hierarchy_clauses]
[attribute_clasues];
```

For an example of the use of the DIMENSION command let's use the PLAN_TABLE used by EXPLAIN PLAN which contains the recursive relationship between ID and PARENT_ID columns.

```
SQL> desc plan_table
 Name                           Null?    Type
 ------------------------------ -------- ------------
 STATEMENT_ID                            VARCHAR2(30)
 TIMESTAMP                               DATE
 REMARKS                                 VARCHAR2(80)
 OPERATION                               VARCHAR2(30)
 OPTIONS                                 VARCHAR2(30)
 OBJECT_NODE                             VARCHAR2(128)
 OBJECT_OWNER                            VARCHAR2(30)
 OBJECT_NAME                             VARCHAR2(30)
 OBJECT_INSTANCE                         NUMBER(38)
 OBJECT_TYPE                             VARCHAR2(30)
 OPTIMIZER                               VARCHAR2(255)
 SEARCH_COLUMNS                          NUMBER
 ID                                      NUMBER(38)
 PARENT_ID                               NUMBER(38)
 POSITION                                NUMBER(38)
 COST                                    NUMBER(38)
 CARDINALITY                             NUMBER(38)
 BYTES                                   NUMBER(38)
 OTHER_TAG                               VARCHAR2(255)
 PARTITION_START                         VARCHAR2(255)
 PARTITION_STOP                          VARCHAR2(255)
 PARTITION_ID                            NUMBER(38)
 OTHER                                   LONG


SQL> create dimension test_dim
  2  level child_id is plan_table.id
  3  level parent_id is plan_table.parent_id
  4  hierarchy plan (child_id child of parent_ID)
  5  attribute parent_id determines plan_table.statement_id
  6  /

Dimension created.

SQL>
```

With the dimension test_dim, if we now created a materialized view on the PLAN_TABLE any queries attempting to do a ROLLUP or CUBE type operation across the ID-PARENT_ID levels would use the connection information stored in the DIMENSION to rewrite the query. The CREATE DIMENSION command also allows forcing of the creation if the tables don't exist or you don't have permission

on them, as well as allowing join conditions to be specified between child and parent levels.

Altering DIMENSION Objects

The ALTER DIMENSION command is used to add or drop LEVEL, HIERARCHY or ATTRIBUTE information for a DIMENSION as well as force a compile of the object. An example would be if the PLAN_TABLE in the CREATE example didn't exist and we had used the FORCE keyword in the command. The views DBA_DIMENSIONS, ALL_DIMENSIONS and USER_DIMENSIONS are used to monitor DIMENSION status, the INVALID (shown as I in the example below) tells the state of the DIMENSION, either Y for an INVALID DIMENSION or N for a VALID DIMENSION.

```
SQL> select * from user_dimensions;

OWNER                          DIMENSION_NAME                 I   REVISION
------------------------------ ------------------------------ - ----------
SYSTEM                         TEST_DIM                       Y          1

SQL> @d:\orant81\rdbms\admin\utlxplan

Table created.

SQL> alter dimension test_dim compile;

Dimension altered.

SQL> select * from user_dimensions;

OWNER                          DIMENSION_NAME                 I   REVISION
------------------------------ ------------------------------ - ----------
SYSTEM                         TEST_DIM                       N          1
```

As was said above, we could also have added or removed levels, hierarchies or attributes using the ALTER DIMENSION command.

Dropping DIMENSIONS

A DIMENSION object is dropped using the DROP DIMENSION command. The syntax of the DROP command is:

```
DROP DIMENSION [schema.]dimension_name;
```

An example is:

```
SQL> DROP DIMENSION system.test_dim;

Dimension dropped.
```

Managing CPU Utilization for Data Warehouses in Oracle8i

In data warehouses diverse groups of users may look at the data warehouse to find information important to their group. While we like to believe everyone is equal, face it, if the CEO wants a report his needs come over and above Joe Clerk's needs for a different report. In earlier releases of Oracle you could use profiles to restrict specific types of resources but this was unwieldy and produced unpredictable results. New in Oracle8i is the concept of Oracle resource groups. A resource group specification allows you to specify that a specific group of database users can only use a certain percentage of the CPU resources on the system. A resource plan must be developed that defines the various levels within the application and their percentage of CPU resources in a waterfall type structure where each subsequent levels percentages are based on the previous levels.

Creating a Resource Plan

Rather than have a simple CREATE RESOURCE PLAN command, Oracle8i has a series of packages which must be run in a specific order to create a proper resource plan. All resource plans are created in a pending area before being validated and committed to the database. The requirements for a valid resource plan are outlined in the definition of the DBMS_RESOURCE_MANAGER.VALIDATE_PENDING_AREA procedure below.

Resource plans can have up to 32 levels with 32 groups per level allowing the most complex resource plan to be easily grouped. Multiple plans, sub-plans and groups can all be tied together into an application spanning CPU resource utilization rule set. This rule set is known as a set of directives. An example would be a simple 2-tier plan like that shown in Figure 1.

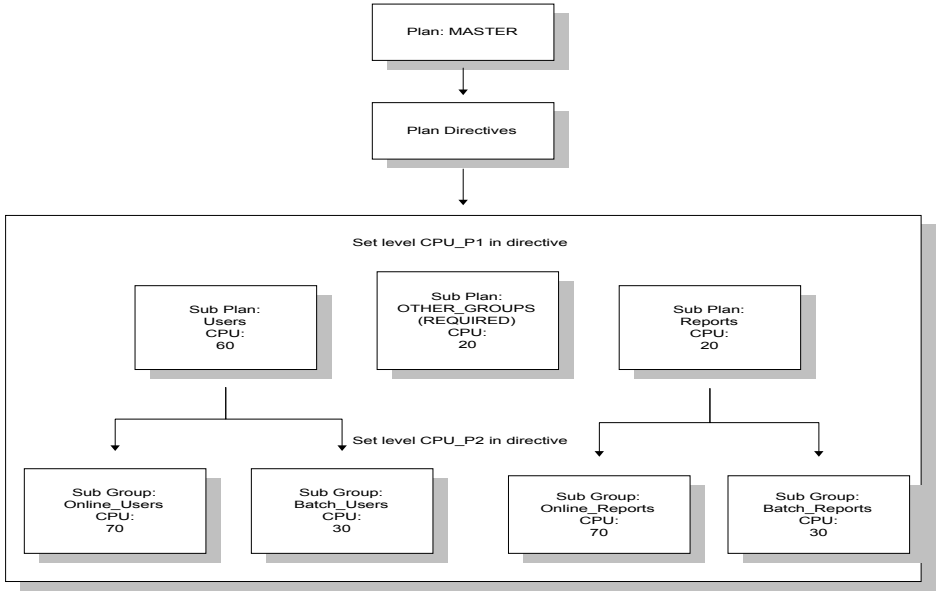

Figure 1 Example Resource Plan

An example of how this portioning out of CPU resources works would be to examine what happens in the plan shown in Figure 1. In figure 1 we have a top level called MASTER which can have up to 100% of the CPU if it requires it. The next level of the plan creates two sub-plans, USERS and REPORTS which will get maximums of 60 and 20 percent of the CPU respectively (we also have the required plan OTHER_GROUPS to which we have assigned 20 percent, if a user is not assigned to a specific group, they get OTHERS). Under USERS we have two groups, ONLINE_USERS and BATCH_USERS.

ONLINE_USERS gets 70 percent of USERS 60 percent or an overall percent of CPU of 42 percent while the other sub-group, BATCH_USERS gets 30 percent of the 60 percent for a total overall percent of 18.

The steps for creating a resource plan, its directives and its groups is shown in Figure 2.

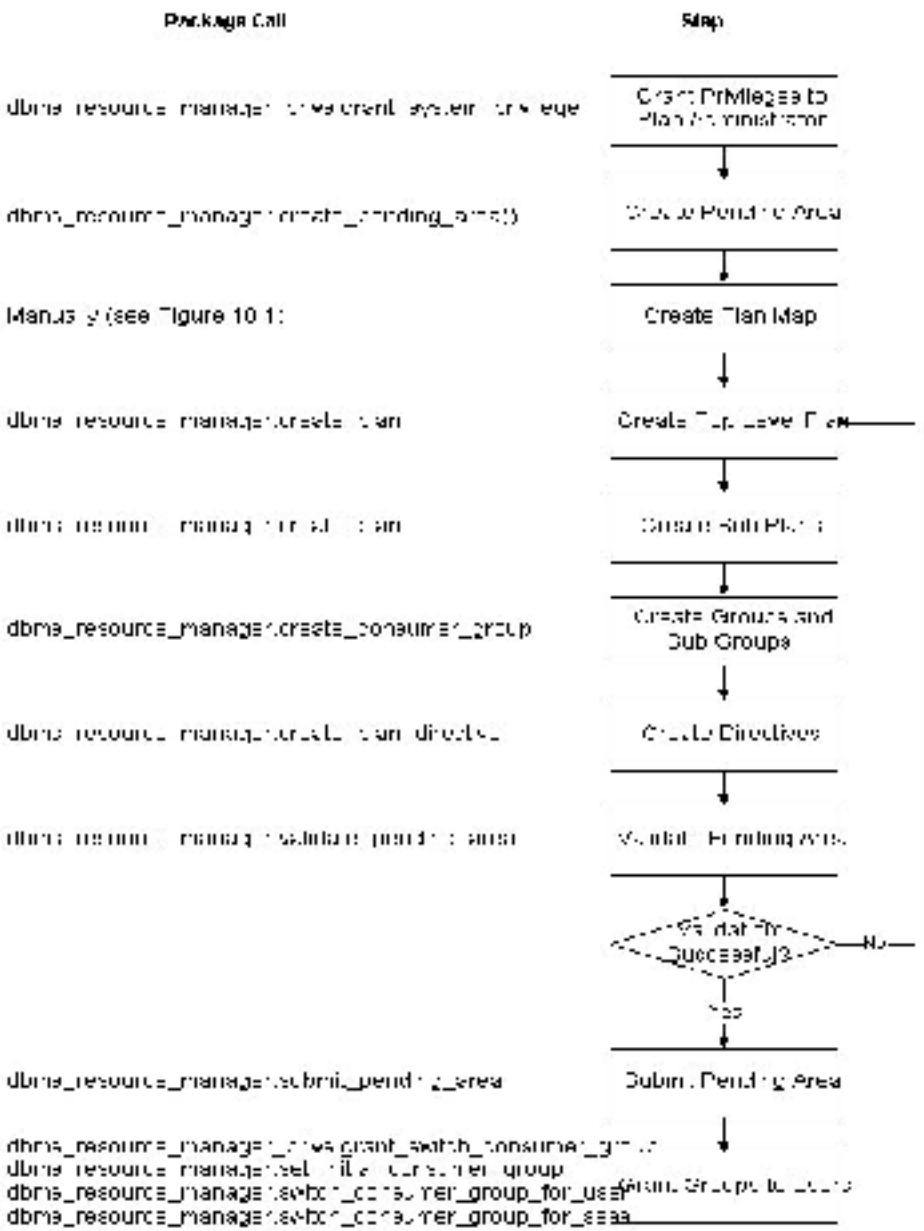

Figure 2 Steps to Create a Resource Plan

One thing to notice about Figure 2 is that the last step shows several possible packages which can be run to assign or change the assignment of resource

groups. The first package listed, DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP must be run the first time a user is assigned to a resource group or you won't be able to assign the user to the group. After the user has been given the SWITCH_CONSUMER_GROUP system privilege you don't have to re-run the package.

Figure 3 shows the code to create the resource plan in Figure 1. Figure 4 shows the results from running the source in figure 3.

```
set echo on
spool test_resource_plan.doc
-- Grant system privilege to plan administrator
--
execute
dbms_resource_manager_privs.grant_system_privilege('SYSTEM','ADMINISTER_RE
SOURCE_MANAGER',TRUE);
--
--connect to plan administrator
--
CONNECT system/system_test@ortest1.world
--
-- Create Plan Pending Area
--
EXECUTE dbms_resource_manager.create_pending_area();
--
-- Create plan
--
execute dbms_resource_manager.create_plan('MASTER','Example Resource
Plan','EMPHASIS');
execute dbms_resource_manager.create_plan('USERS','Example Resource Sub
Plan','EMPHASIS');
execute dbms_resource_manager.create_plan('REPORTS','Example Resource Sub
Plan','EMPHASIS');
--
--Create tiers of groups in plan
--
EXECUTE dbms_resource_manager.create_consumer_group('ONLINE_USERS','3rd
level group','ROUND-ROBIN');
EXECUTE dbms_resource_manager.create_consumer_group('BATCH_USERS','3rd
level group','ROUND-ROBIN');
EXECUTE dbms_resource_manager.create_consumer_group('ONLINE_REPORTS','2rd
level group','ROUND-ROBIN');
EXECUTE dbms_resource_manager.create_consumer_group('BATCH_REPORTS','2rd
level group','ROUND-ROBIN');
--
-- Create plan directives
--
EXECUTE dbms_resource_manager.create_plan_directive('MASTER', 'USERS',
0,60,0,0,0,0,0,0,NULL);
EXECUTE dbms_resource_manager.create_plan_directive('MASTER', 'REPORTS',
0,20,0,0,0,0,0,0,NULL);
EXECUTE
dbms_resource_manager.create_plan_directive('MASTER','OTHER_GROUPS',
0,20,0,0,0,0,0,0,NULL);
```

```
EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'ONLINE_USERS',  0,0,70,0,0,0,0,0,NULL);
EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'BATCH_USERS',   0,0,30,0,0,0,0,0,NULL);
EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','ONLINE_REPORTS',0,0
,70,0,0,0,0,0,NULL);
EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','BATCH_REPORTS',
0,0,30,0,0,0,0,0,NULL);
--
-- Verify Plan
--
EXECUTE dbms_resource_manager.validate_pending_area;
--
-- Submit Plan
--
EXECUTE dbms_resource_manager.submit_pending_area;
spool off
set echo off
```

Figure 3 Script to create example resource plan

Notice how the script in figure 3 follows the chart in Figure 2. These are the proper steps to create a resource plan. Figure 4 shows the results from running the script in Figure 3.

```
SQL> -- Grant system privilege to plan administrator
SQL> --
SQL> execute
dbms_resource_manager_privs.grant_system_privilege('SYSTEM','ADMINISTER_RE
SOURCE_MANAGER',TRUE);

PL/SQL procedure successfully completed.

SQL> --
SQL> --connect to plan administrator
SQL> --
SQL> CONNECT system/system_test@ortest1.world
Connected.
SQL> --
SQL> -- Create Plan Pending Area
SQL> --
SQL> EXECUTE dbms_resource_manager.create_pending_area();

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Create plan
SQL> --
SQL> execute dbms_resource_manager.create_plan('MASTER','Example Resource
Plan','EMPHASIS');

PL/SQL procedure successfully completed.
```

```
SQL> execute dbms_resource_manager.create_plan('USERS','Example Resource
Sub Plan','EMPHASIS');

PL/SQL procedure successfully completed.

SQL> execute dbms_resource_manager.create_plan('REPORTS','Example Resource
Sub Plan','EMPHASIS');

PL/SQL procedure successfully completed.

SQL> --
SQL> --Create tiers of groups in plan
SQL> --
SQL> EXECUTE
dbms_resource_manager.create_consumer_group('ONLINE_USERS','3rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_consumer_group('BATCH_USERS','3rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_consumer_group('ONLINE_REPORTS','2rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_consumer_group('BATCH_REPORTS','2rd level
group','ROUND-ROBIN');

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Create plan directives
SQL> --
SQL> EXECUTE dbms_resource_manager.create_plan_directive('MASTER',
'USERS',           0,60,0,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_resource_manager.create_plan_directive('MASTER',
'REPORTS',         0,20,0,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_plan_directive('MASTER','OTHER_GROUPS',
0,20,0,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'ONLINE_USERS',  0,0,70,0,0,0,0,0,NULL);
```

```
PL/SQL procedure successfully completed.

SQL> EXECUTE dbms_resource_manager.create_plan_directive('USERS',
'BATCH_USERS',   0,0,30,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','ONLINE_REPORTS',0,0
,70,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> EXECUTE
dbms_resource_manager.create_plan_directive('REPORTS','BATCH_REPORTS',
0,0,30,0,0,0,0,0,NULL);

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Verify Plan
SQL> --
SQL> EXECUTE dbms_resource_manager.validate_pending_area;

PL/SQL procedure successfully completed.

SQL> --
SQL> -- Submit Plan
SQL> --
SQL> EXECUTE dbms_resource_manager.submit_pending_area;

PL/SQL procedure successfully completed.

SQL> spool off
```

Figure 4 Example run of script to create example resource plan

The other operations allowed against the components of the resource plan are alter and drop. Let's look at a quick drop example in Figure 5.

```
EXECUTE dbms_resource_manager.delete_plan('MASTER');
EXECUTE dbms_resource_manager.delete_plan('USERS');
EXECUTE dbms_resource_manager.delete_plan('REPORTS');
--
--delete tiers of groups in plan
--
EXECUTE dbms_resource_manager.delete_consumer_group('ONLINE_USERS');
EXECUTE dbms_resource_manager.delete_consumer_group('BATCH_USERS');
EXECUTE dbms_resource_manager.delete_consumer_group('ONLINE_REPORTS');
EXECUTE dbms_resource_manager.delete_consumer_group('BATCH_REPORTS');
```

Figure 5 Example Drop Procedure

Notice how you must drop all parts of the plan, this is because Oracle allows Orphan groups and plans to exist. As you can tell from looking at the scripts the DBMS_RESOURCE_MANAGER and DBMS_RESOURCE_MANAGER_PRIVS packages are critical to implementing Oracle resource groups.

Let's examine these packages.

DBMS_RESOURCE_MANAGER Package

The DBMS_RESOURCE_MANAGER package is used to administer the new resource plan and consumer group options in Oracle8i. The package contains several procedures that are used to create, modify, drop and grant access to resource plans, groups, directives and pending areas. The invoker must have the ADMINISTER_RESOURCE_MANAGER system privilege to execute these procedures. The procedures to grant and revoke this privilege are in the package DBMS_RESOURCE_MANAGER_PRIVS. The procedures in DBMS_RESOURCE_MANAGER are listed in table 1.

Procedure	Purpose
CREATE_PLAN	Creates entries which define resource plans.
UPDATE_PLAN	Updates entries which define resource plans.
DELETE_PLAN	Deletes the specified plan as well as all the plan directives it refers to.
DELETE_PLAN_CASCADE	Deletes the specified plan as well as all its descendants (plan directives, subplans, consumer groups).
CREATE_CONSUMER_ GROUP	Creates entries which define resource consumer groups.
UPDATE_CONSUMER_ GROUP	Updates entries which define resource consumer groups.
DELETE_CONSUMER_ GROUP	Deletes entries which define resource consumer groups.
CREATE_PLAN_DIRECTIVE	Creates resource plan directives.
UPDATE_PLAN_DIRECTIVE	Updates resource plan directives.
DELETE_PLAN_DIRECTIVE	Deletes resource plan directives.
CREATE_PENDING_AREA	Creates a work area for changes to resource manager objects.
VALIDATE_PENDING_AREA	Validates pending changes for the resource manager.
CLEAR_PENDING_AREA	Clears the work area for the resource manager.
SUBMIT_PENDING_AREA	Submits pending changes for the resource manager.
SET_INITIAL_CONSUMER_ GROUP	Assigns the initial resource consumer group for a user.

Procedure	Purpose
SWITCH_CONSUMER_ GROUP_FOR_SESS	Changes the resource consumer group of a specific session.
SWITCH_CONSUMER_ GROUP_FOR_USER	Changes the resource consumer group for all sessions with a given user name.

Table 1 DBMS_RESOURCE_MANAGER_PACKAGES

DBMS_RESOURCE_MANGER Procedure Syntax

The calling syntax for all of the DBMS_RESOURCE_MANAGER packages follow.

Syntax for the CREATE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN (
   plan                       IN VARCHAR2,
   comment                    IN VARCHAR2,
   cpu_mth                    IN VARCHAR2 DEFAULT 'EMPHASIS',
   max_active_sess_target_mth IN VARCHAR2 DEFAULT
                                 'MAX_ACTIVE_SESS_ABSOLUTE',
   parallel_degree_limit_mth  IN VARCHAR2 DEFAULT
                            'PARALLEL_DEGREE_LIMIT_ABSOLUTE');
```

Where:

- Plan - the plan name
- Comment - any text comment you want associated with the plan name
- Cpu_mth - one of EMPHASIS or ROUND-ROBIN
- max_active_sess_target_mth - allocation method for max. active sessions
- parallel_degree_limit_mth - allocation method for degree of parallelism

Syntax for the UPDATE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN (
   plan                           IN VARCHAR2,
   new_comment                    IN VARCHAR2 DEFAULT NULL,
   new_cpu_mth                    IN VARCHAR2 DEFAULT NULL,
   new_max_active_sess_target_mth IN VARCHAR2 DEFAULT NULL,
   new_parallel_degree_limit_mth  IN VARCHAR2 DEFAULT NULL);
```

Where:

- plan - name of resource plan

- new_comment - new user's comment
- new_cpu_mth - name of new allocation method for CPU resources
- new_max_active_sess_target_mth - name of new method for max. active sessions
- new_parallel_degree_limit_mth - name of new method for degree of parallelism

Syntax for the DELETE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN (
   plan IN VARCHAR2);
```

Where:

- Plan - Name of resource plan to delete.

Syntax for the DELETE_PLAN Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_CASCADE (
   plan IN VARCHAR2);
```

Where:

- Plan - Name of plan.

Syntax for the CREATE_RESOURCE_GROUP Procedure:

```
DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP (
   consumer_group IN VARCHAR2,
   comment        IN VARCHAR2,
   cpu_mth        IN VARCHAR2 DEFAULT 'ROUND-ROBIN');
```

Where:

- consumer_group - Name of consumer group.
- Comment - User's comment.
- cpu_mth - Name of CPU resource allocation method.

Syntax for the UPDATE_RESOURCE_GROUP Procedure:

```
DBMS_RESOURCE_MANAGER.UPDATE_CONSUMER_GROUP (
   consumer_group IN VARCHAR2,
   new_comment    IN VARCHAR2 DEFAULT NULL,
   new_cpu_mth    IN VARCHAR2 DEFAULT NULL);
```

Where:

- plan - name of resource plan
- new_comment - new user's comment
- new_cpu_mth - name of new allocation method for CPU resources
- new_max_active_sess_target_mth - name of new method for max. active sessions
- new_parallel_degree_limit_mth - name of new method for degree of parallelism

Syntax for the DELTE_RESOURCE_GROUP Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_CONSUMER_GROUP (
   consumer_group IN VARCHAR2);
```

Where:

- plan - name of resource plan.

Syntax for the CREATE_PLAN_DIRECTIVE Procedure:

```
DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE (
   plan                     IN VARCHAR2,
   group_or_subplan         IN VARCHAR2,
   comment                  IN VARCHAR2,
   cpu_p1                   IN NUMBER   DEFAULT NULL,
   cpu_p2                   IN NUMBER   DEFAULT NULL,
   cpu_p3                   IN NUMBER   DEFAULT NULL,
   cpu_p4                   IN NUMBER   DEFAULT NULL,
   cpu_p5                   IN NUMBER   DEFAULT NULL,
   cpu_p6                   IN NUMBER   DEFAULT NULL,
   cpu_p7                   IN NUMBER   DEFAULT NULL,
   cpu_p8                   IN NUMBER   DEFAULT NULL,
   max_active_sess_target_p1 IN NUMBER   DEFAULT NULL,
   parallel_degree_limit_p1  IN NUMBER   DEFAULT NULL);
```

Where:

- plan - name of resource plan
- group_or_subplan - name of consumer group or subplan

- comment - comment for the plan directive
- cpu_p1 - first parameter for the CPU resource allocation method
- cpu_p2 - second parameter for the CPU resource allocation method
- cpu_p3 - third parameter for the CPU resource allocation method
- cpu_p4 - fourth parameter for the CPU resource allocation method
- cpu_p5 - fifth parameter for the CPU resource allocation method
- cpu_p6 - sixth parameter for the CPU resource allocation method
- cpu_p7 - seventh parameter for the CPU resource allocation method
- cpu_p8 - eighth parameter for the CPU resource allocation method
- max_active_sess_target_p1 - first parameter for the max. active sessions allocation method
- (RESERVED FOR FUTURE USE)
- parallel_degree_limit_p1 - first parameter for the degree of parallelism allocation method

Syntax for the UPDATE_PLAN_DIRECTIVE Procedure:

```
DBMS_RESOURCE_MANAGER.UPDATE_PLAN_DIRECTIVE (
   plan                          IN VARCHAR2,
   group_or_subplan              IN VARCHAR2,
   new_comment                   IN VARCHAR2 DEFAULT NULL,
   new_cpu_p1                    IN NUMBER   DEFAULT NULL,
   new_cpu_p2                    IN NUMBER   DEFAULT NULL,
   new_cpu_p3                    IN NUMBER   DEFAULT NULL,
   new_cpu_p4                    IN NUMBER   DEFAULT NULL,
   new_cpu_p5                    IN NUMBER   DEFAULT NULL,
   new_cpu_p6                    IN NUMBER   DEFAULT NULL,
   new_cpu_p7                    IN NUMBER   DEFAULT NULL,
   new_cpu_p8                    IN NUMBER   DEFAULT NULL,
   new_max_active_sess_target_p1 IN NUMBER   DEFAULT NULL,
   new_parallel_degree_limit_p1  IN NUMBER   DEFAULT NULL);
```

Where:

- plan - name of resource plan
- group_or_subplan - name of group or subplan
- new_comment - comment for the plan directive
- new_cpu_p1 - first parameter for the CPU allocation method
- new_cpu_p2 - parameter for the CPU allocation method

- new_cpu_p3- parameter for the CPU allocation method
- new_cpu_p4 - parameter for the CPU allocation method
- new_cpu_p5 - parameter for the CPU allocation method
- new_cpu_p6 - parameter for the CPU allocation method
- new_cpu_p7 - parameter for the CPU allocation method
- new_cpu_p8 - parameter for the CPU allocation method
- new_max_active_sess_target_p1 - first parameter for the max. active sessions allocation method
- (RESERVED FOR FUTURE USE)
- new_parallel_degree_limit_p1 - first parameter for the degree of parallelism allocation method

Syntax for the DELETE_PLAN_DIRECTIVE Procedure:

```
DBMS_RESOURCE_MANAGER.DELETE_PLAN_DIRECTIVE (
   plan             IN VARCHAR2,
   group_or_subplan IN VARCHAR2);
```

Where:

- plan - name of resource plan
- group_or_subplan - name of group or subplan.

Syntax for CREATE_PENDING_AREA Procedure:

This procedure lets you make changes to resource manager objects.

All changes to the plan schema must be done within a pending area. The pending area can be thought of as a "scratch" area for plan schema changes. The administrator creates this pending area, makes changes as necessary, possibly validates these changes, and only when the submit is completed do these changes become active.

You may, at any time while the pending area is active, view the current plan schema with your changes by selecting from the appropriate user views.

At any time, you may clear the pending area if you want to stop the current changes. You may also call the VALIDATE procedure to confirm whether the changes you has made are valid. You do not have to do your changes in a given

order to maintain a consistent group of entries. These checks are also implicitly done when the pending area is submitted.

Note: Oracle allows "orphan" consumer groups (i.e., consumer groups that have no plan directives that refer to them). This is in anticipation that an administrator may want to create a consumer group that is not currently being used, but will be used in the future. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.CREATE_PENDING_AREA;
```

Syntax of the VALIDATE_PENDING_AREA Procedure:

The VALIDATE_PENDING_AREA procedure is used to validate the contents of a pending area before they are submitted. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.VALIDATE_PENDING_AREA;
```

Usage Notes For the Validate and Submit Procedures:

The following rules must be adhered to, and they are checked whenever the validate or submit procedures are executed:

1. No plan schema may contain any loops.
2. All plans and consumer groups referred to by plan directives must exist.
3. All plans must have plan directives that refer to either plans or consumer groups.
4. All percentages in any given level must not add up to greater than 100 for the emphasis resource allocation method.
5. No plan may be deleted that is currently being used as a top plan by an active instance.
6. For Oracle8i, the plan directive parameter, parallel_degree_limit_p1, may only appear in plan directives that refer to consumer groups (i.e., not at subplans).
7. There cannot be more than 32 plan directives coming from any given plan (i.e., no plan can have more than 32 children).
8. There cannot be more than 32 consumer groups in any active plan schema.
9. Plans and consumer groups use the same namespace; therefore, no plan can have the same name as any consumer group.

10. There must be a plan directive for OTHER_GROUPS somewhere in any active plan schema.This ensures that a session not covered by the currently active plan is allocated resources as specified by the OTHER_GROUPS directive.

If any of the above rules are broken when checked by the VALIDATE or SUBMIT procedures, then an informative error message is returned. You may then make changes to fix the problem(s) and reissue the validate or submit procedures.

Syntax of the CLEAR_PENDING_AREA Procedure:

The CLEAR_PENDING_AREA procedure clears the pending area without submitting it, all changes or entries are lost. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.CLEAR_PENDING_AREA;
```

Syntax of the SUBMIT_PENDING_AREA Procedure:

The SUBMIT_PENDING_AREA procedure submits the contents of the pending area. First the contents are validated and then they are stored as valid in the database. The procedure has no arguments.

```
DBMS_RESOURCE_MANAGER.SUBMIT_PENDING_AREA;
```

Syntax of the SET_ INITIAL_CONSUMER_GROUP Procedure:

The SET_INITIAL_CONSUMER_GROUP procedure sets the initial consumer group to which a user will belong. The user must have been granted SWITCH_RESOURCE_GROUP permission before you attempt to run this procedure.

```
DBMS_RESOURCE_MANAGER.SET_INITIAL_CONSUMER_GROUP (
   user           IN VARCHAR2,
   consumer_group IN VARCHAR2);
```

Where:

- User – The user that is to have the resource group set.
- Consumer_group – The resource (or consumer) group to grant to the user.

Syntax of the SWITCH_CONSUMER_GROUP_FOR_SESS Procedure:

The SWITCH_RESOURCE_GROUP_FOR_SESS procedure allows an administrator to switch a user's consumer group for the duration of the current session.

```
DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_SESS(
  SESSION_ID      IN NUMBER,
  SESSION_SERIAL  IN NUMBER,
  CONSUMER_GROUP  IN VARCHAR2);
```

Where:

- session_id - SID column from the view V$SESSION
- session_serial - SERIAL# column from the view V$SESSION
- consumer_group - name of the consumer group of which to switch.

Syntax of the SWITCH_CONSUMER_GROUP_FOR_USER Procedure:

The SWITCH_CONSUMER_GROUP_FOR_USER switches a user's default consumer group to a new group. This is a permanent change.

```
DBMS_RESOURCE_MANAGER.SWITCH_CONSUMER_GROUP_FOR_USER (
   user           IN VARCHAR2,
   consumer_group IN VARCHAR2);
```

Where:

- user - name of the user
- consumer_group - name of the consumer group to switch to

DBMS_RESOURCE_MANAGER_PRIVS Package

The DBMS_RESOURCE_MANAGER package has a companion package that grants privileges in the realm of the resource consumer option. The companion package is DBMS_RESOURCE_MANAGER_PRIVS. The procedures inside DBMS_RESOURCE_MANAGER_PRIVS are documented in table 2.

Procedure	Purpose
GRANT_SYSTEM_PRIVILEGE	Performs a grant of a system privilege.
REVOKE_SYSTEM_ PRIVILEGE	Performs a revoke of a system privilege.

Procedure	Purpose
GRANT_SWITCH_ CONSUMER_GROUP	Grants the privilege to switch to resource consumer groups.
REVOKE_SWITCH_ CONSUMER_GROUP	Revokes the privilege to switch to resource consumer groups.

Table 2 DBMS_RESOURCE_MANAGER_PRIVS Procedures

DBMS_RESOURCE_MANGER_PRIVS Procedure Syntax

The calling syntax for all of the DBMS_RESOURCE_MANAGER_PRIVS packages follow.

Syntax for the GRANT_SYSTEM_PRIVILEGE Procedure:

The GRANT_SYSTEM_PRIVILEGE procedure grants ADMINISTER_RESOURCE_MANAGER privilege to a user. Currently there is only one resource group system grant.

```
DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SYSTEM_PRIVILEGE (
   grantee_name   IN VARCHAR2,
   privilege_name IN VARCHAR2 DEFAULT 'ADMINISTER_RESOURCE_MANAGER',
   admin_option   IN BOOLEAN);
```

Where:

- grantee_name - Name of the user or role to whom privilege is to be granted.
- privilege_name - Name of the privilege to be granted.
- admin_option - TRUE if the grant is with admin_option, FALSE otherwise.

Syntax for the REVOKE_SYSTEM_PRIVILGE Procedure:

The REVOKE_SYSTEM_PRIVILEGE procedure revokes the ADMINISTER_RESOURCE_MANAGER privilege from a user.

```
DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SYSTEM_PRIVILEGE (
   revokee_name   IN VARCHAR2,
   privilege_name IN VARCHAR2 DEFAULT 'ADMINISTER_RESOURCE_MANAGER');
```

Where:

- revokee_name - Name of the user or role from whom privilege is to be revoked.
- privilege_name - Name of the privilege to be revoked.

Syntax of the GRANT_SWITCH_CONSUMER_GROUP Procedure:

The GRANT_SWITCH_CONSUMER_GROUP procedure grants a user the ability to switch resource groups. This privilege must be granted to a user before their initial resource group can be granted.

```
DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP (
   grantee_name   IN VARCHAR2,
   consumer_group IN VARCHAR2,
   grant_option   IN BOOLEAN);
```

Where:

- grantee_name - Name of the user or role to whom privilege is to be granted.
- consumer_group - Name of consumer group.
- grant_option - TRUE if grantee should be allowed to grant access, FALSE otherwise.

Usage Notes

1. If you grant permission to switch to a particular consumer group to a user, then that user can immediately switch their current consumer group to the new consumer group.
2. If you grant permission to switch to a particular consumer group to a role, then any users who have been granted that role and have enabled that role can immediately switch their current consumer group to the new consumer group.
3. If you grant permission to switch to a particular consumer group to PUBLIC, then any user can switch to that consumer group.
4. If the grant_option parameter is TRUE, then users granted switch privilege for the consumer group may also grant switch privileges for that consumer group to others.
5. In order to set the initial consumer group of a user, you must grant the switch privilege for that group to the user.

Syntax of the REVOKE_SWITCH_CONSUMER_GROUP Procedure:

The REVOKE_SWITCH_CONSUMER_GROUP procedure revokes the ability of a user to switch their resource group.

```
DBMS_RESOURCE_MANAGER_PRIVS.REVOKE_SWITCH_CONSUMER_GROUP (
   revokee_name   IN VARCHAR2,
   consumer_group IN VARCHAR2);
```

Where:

- revokee_name - Name of user/role from which to revoke access.
- consumer_group - Name of consumer group.

Usage Notes

1. If you revoke a user's switch privilege for a particular consumer group, then any subsequent attempts by that user to switch to that consumer group will fail.
2. If you revoke the initial consumer group from a user, then that user will automatically be part of the DEFAULT_CONSUMER_GROUP (OTHERS) consumer group when logging in.
3. If you revoke the switch privilege for a consumer group from a role, then any users who only had switch privilege for the consumer group via that role will not be subsequently able to switch to that consumer group.
4. If you revoke the switch privilege for a consumer group from PUBLIC, then any users who could previously only use the consumer group via PUBLIC will not be subsequently able to switch to that consumer group.

Summary

By carefully planning your resource allocation into plans and resource groups a multi-tier resource allocation plan can be quickly developed. By allocating CPU resources you can be sure that processing power is concentrated where it needs to be such that the CEO isn't waiting on a sub-clerk's process to finish before they get their results.

This lesson has shown how to use the various DBMS packages to configure and maintain a resource plan with its associated consumer groups.

This lesson is an excerpt from the upcoming book: "Oracle8i Administration and Management", Michael R. Ault, John Wiley and Sons publishing with permission.

Restricting Access by Rows in an Oracle8i Data Warehouse

New to Oracle8i is the concept of row level access restriction. For years DBAs have requested some form of conditional grant were access to specific rows can be easily restricted or granted based on user or group membership. Oracle has finally given DBAs the functionality of conditional grants in the form of row level security. In a data warehouse there may be data that is restricted in nature, the pay for a particular department, the locations of specific assets, etc. The new row level security, since it is restricted at the database level, prohibits access to restricted rows even when ad hoc tools are used to query the warehouse.

Row level security is managed using a combination of Oracle8i contexts, stored procedures, database level triggers and the DBMS_RLS package. The entire row level security concept is tightly bound to the concept of a database policy. Generally speaking a policy will require:

1. a context
2. a procedure to implement the context
3. a database (Oracle8i) level trigger that monitors login activity
4. a security procedure to implement the policy
5. a policy declaration

Row level security control depends on certain environment variables, know as contexts, to be set. The DBMS_CONTEXT package is used to set the various context variables used by the RLS policy.

Figure 1 shows a flowchart of how to implement a simple security policy.

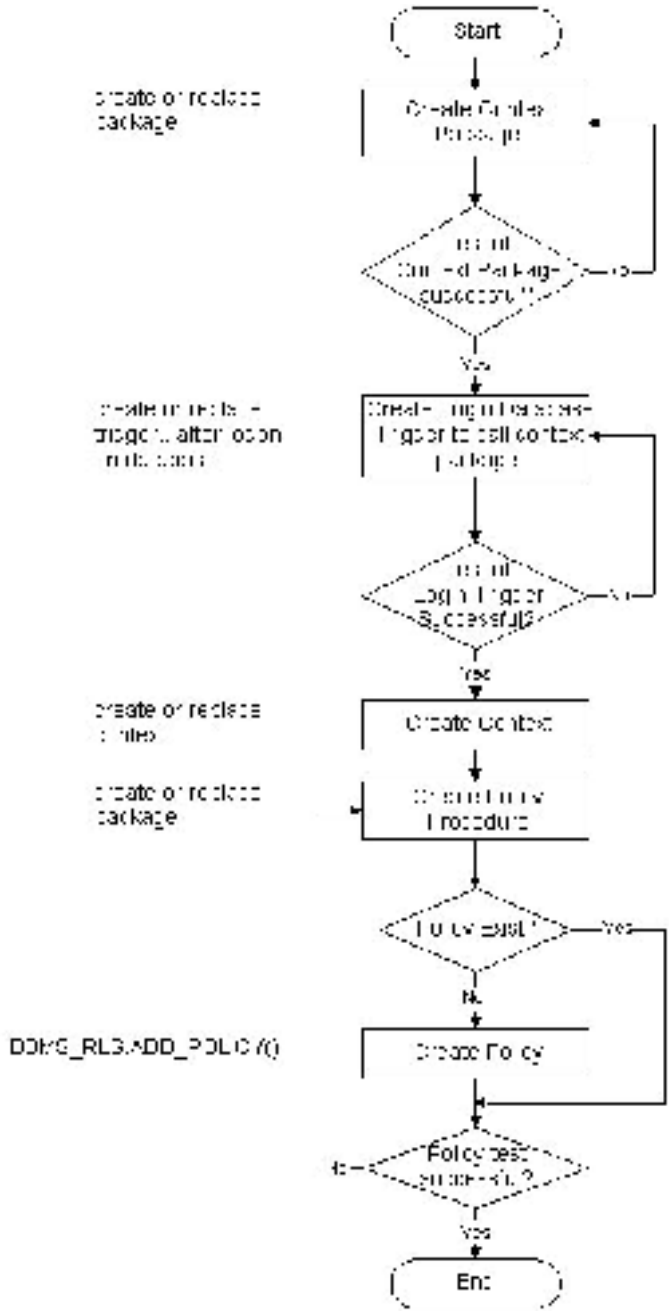

Figure 1: Steps to Implement a security policy

As you can see the process is not very complex. Let's examine each step and see what is really involved.

In the first step a context package or procedure is developed which will then be used by a login trigger to set each users context variables. This step is vital in that if the context variables aren't set it is many times more difficult to implement row level security using the DBMS_RLS package. The package or procedure used to set the context variables should resemble the one shown in figure 2.

```
CREATE OR REPLACE PACKAGE graphics_app AUTHID DEFINER AS
PROCEDURE get_graphics_function(usern IN VARCHAR2, graphics_function OUT
VARCHAR2);
PROCEDURE set_graphics_context(usern IN VARCHAR2);
END;
/
SET ARRAYSIZE 1
SHO ERR
CREATE OR REPLACE PACKAGE BODY graphics_app AS
graphics_user VARCHAR2(32);
graphics_function VARCHAR2(32);
PROCEDURE get_graphics_function(usern IN VARCHAR2, graphics_function OUT
VARCHAR2) IS
BEGIN
SELECT user_function INTO graphics_function FROM
graphics_dba.graphics_users
WHERE username=usern;
END get_graphics_function;
PROCEDURE set_graphics_context(usern IN VARCHAR2) IS
BEGIN
graphics_app.get_graphics_function(usern,graphics_function);
DBMS_SESSION.SET_CONTEXT('GRAPHICS_SEC','GRAPHICS_FUNCTION',graphics_funct
ion);
DBMS_SESSION.SET_CONTEXT('GRAPHICS_SEC','GRAPHICS_USER',usern);
END set_graphics_context;
END graphics_app;
/
SHOW ERR
```

Figure 2: Example Context Setting Procedure

In the package in figure 2 are two procedures, one that retrieves a users graphics function from a pre-built and populated table (GET_GRAPHICS_FUNCTION) and the other which is used to set the users context variables based on using the DBMS_SESSION.SET_CONTEXT procedure provided by Oracle (SET_GRAPHICS_CONTEXT).

Of course the procedures in figure 2 wouldn't be much use without a trigger that could run the procedure whenever a user logged on the system. Until Oracle8i this would have involved setting auditing on for login, moving the aud$ table from SYS ownership and setting the ownership to another user, resetting all of the synonyms pointing to aud$ and then building an on-insert trigger to perform the

actual work. In Oracle8i all we have to do is build a database level trigger similar to the one shown in figure 3.

```
CREATE OR REPLACE TRIGGER set_graphics_context AFTER LOGON ON DATABASE
DECLARE
username VARCHAR2(30);
BEGIN
  username:=SYS_CONTEXT('USERENV','SESSION_USER');
  graphics_app.set_graphics_context(username);
  EXCEPTION
   WHEN OTHERS THEN
   NULL;
END;
/
```

Figure 3: Example Database Logon Trigger

Once we have an operating context setting package and a database login trigger we can proceed to create the required context checking package and the context it checks. Figure 4 shows an example context checking package.

```
CREATE OR REPLACE PACKAGE graphics_sec AUTHID DEFINER AS
FUNCTION graphics_check(obj_schema VARCHAR2, obj_name VARCHAR2)
 RETURN VARCHAR2;
PRAGMA RESTRICT_REFERENCES(GRAPHICS_CHECK,WNDS);
END;
/
SET ARRAYSIZE 1
SHOW ERR
CREATE OR REPLACE PACKAGE BODY graphics_sec AS
FUNCTION graphics_check(obj_schema VARCHAR2, obj_name VARCHAR2)
 RETURN VARCHAR2 AS
d_predicate VARCHAR2(2000);
user_context VARCHAR2(32);
BEGIN
   user_context:=SYS_CONTEXT('graphics_sec','graphics_function');
   IF user_context = 'ADMIN' THEN
     d_predicate:=' 1=1';
dbms_output.put_line(d_predicate);
   ELSIF user_context = 'GENERAL USER' THEN
     d_predicate:=' graphics_usage='||chr(39)||'UNRESTRICTED'||chr(39);
dbms_output.put_line(d_predicate);
   ELSIF user_context='DEVELOPER' THEN
     d_predicate:=' 1=1';
dbms_output.put_line(d_predicate);
   ELSIF user_context IS NULL THEN
     d_predicate:='1=2';
   END IF;
   RETURN d_predicate;
END graphics_check;
END;
/
SHOW ERR
```

Figure 4: Example Context Package

The entire purpose of the package in figure 4 is to return a d_predicate value based on a users graphics_function context value. The d_predicate value is appended to whatever WHERE clause is included with their command, or is appended as a WHERE clause whenever there is no pre-existing clause.

The creation of our graphics security context is rather simple once we have finished the preliminary work, it boils down to one command:

```
CREATE OR REPLACE CONTEXT graphics_sec USING sys.graphics_app;
```

The final step is to set the policy into the database. This is done with the DBMS_RLS package using the procedure ADD_POLICY:

```
BEGIN
  dbms_rls.add_policy(
  'GRAPHICS_DBA','INTERNAL_GRAPHICS','GRAPHICS_POLICY',
  'GRAPHICS_DBA','GRAPHICS_SEC.GRAPHICS_CHECK',
  'SELECT,INSERT,UPDATE,DELETE');
END;
```

The above policy simply ties the components we previously defined into a coherent entity called GRAPHICS_POLICY and implements this policy against the table INTERNAL_GRAPHICS which is in the schema GRAPHICS_DBA. The policy GRACPHICS_POLICY is owned by GRAPHICS_DBA and uses the procedure GRAPHICS_SEC.GRAPHICS_CHECK to verify users can perform SELECT, INSERT, UPDATE and DELETE operations.

The table graphics_users is required in the above example. The table contains the username and their graphics function.

Policy Usage

Policy usage is controlled internally by the Oracle system and adheres to the following usage guidelines:

- SYS user is not restricted by any security policy.
- The policy functions which generate dynamic predicates are called by the server. The following is the required structure for the function:

  ```
  FUNCTION policy_function (object_schema IN VARCHAR2, object_name
  VARCHAR2)
              RETURN VARCHAR2
  ```

  Where:

  - object_schema is the schema owning the table of view.

- object_name is the name of table of view that the policy will apply.

- The maximum length of the predicate that the policy function can return is 2,000 bytes.
- The policy functions must have the purity level of WNDS (write no database state).
- Dynamic predicates generated out of different policies for the same object have the combined effect of a conjunction (ANDed) of all the predicates.
- The security check and object lookup are performed against the owner of the policy function for objects in the subqueries of the dynamic predicates.
- If the function returns a zero length predicate, then it is interpreted as no restriction being applied to the current user for the policy.
- When table alias is required (e.g., parent object is a type table) in the predicate, the name of the table or view itself must be used as the name of the alias. The server constructs the transient view as something like "select c1, c2, ... from tab where <predicate>".
- The checking of the validity of the function is done at runtime for ease of installation and other dependency issues import/export.

DBMS_RLS Package

The entire concept of row level security is based on the use of policies stored in the database. The only way to store policies in the database is to use the DBMS_RLS package.

The DBMS_RLS procedures cause current DML transactions, if any, to commit before the operation. However, the procedures do not cause a commit first if they are inside a DDL event trigger. With DDL transactions, the DBMS_RLS procedures are part of the DDL transaction.

For example, you may create a trigger for CREATE TABLE. Inside the trigger, you may add a column through ALTER TABLE, and you can add a policy through DBMS_RLS. All these operations are in the same transaction as CREATE TABLE, even though each one is a DDL statement. The CREATE TABLE succeeds only if the trigger is completed successfully.

The DBMS_RLS package has the procedures shown in Table 1.

Procedure	Purpose
ADD_POLICY	Creates a fine-grained access control policy to a table or view.
DROP_POLICY	Drops a fine-grained access control policy from a table or view.
REFRESH_POLICY	Causes all the cached statements associated with the policy to be re-parsed.
ENABLE_POLICY	Enables or disables a fine-grained access control policy.

Table 1 Procedure in DBMS_RLS Package

The syntax for calling the DBMS_RLS procedures are shown in the next sections.

Syntax for the ADD_POLICY Procedure:

```
DBMS_RLS.ADD_POLICY (
   object_schema   IN VARCHAR2 := NULL,
   object_name     IN VARCHAR2,
   policy_name     IN VARCHAR2,
   function_schema IN VARCHAR2 := NULL,
   policy_function IN VARCHAR2,
   statement_types IN VARCHAR2 := NULL,
   update_check    IN BOOLEAN  := FALSE,
   enable          IN BOOLEAN  := TRUE);
```

Where:

- object_schema - schema owning the table/view, current user if NULL
- object_name - name of table or view
- policy_name - name of policy to be added
- function_schema - schema of the policy function, current user if NULL
- policy_function - function to generate predicates for this policy
- statement_types - statement type that the policy apply, default is any
- update_check - policy checked against updated or inserted value?
- enable - policy is enabled?

Syntax for the DROP_POLICY Procedure:

```
DBMS_RLS.DROP_POLICY (
   object_schema IN VARCHAR2 := NULL,
   object_name   IN VARCHAR2,
   policy_name   IN VARCHAR2);
```

Where:

- object_schema - Schema containing the table or view (logon user if NULL).
- object_name - Name of table or view.
- policy_name - Name of policy to be dropped from the table or view.

Syntax for the REFRESH_POLICY Procedure:

```
DBMS_RLS.REFRESH_POLICY (
   object_schema IN VARCHAR2 := NULL,
   object_name   IN VARCHAR2 := NULL,
   policy_name   IN VARCHAR2 := NULL);
```

Where:

- object_schema - Schema containing the table or view.
- object_name - Name of table or view that the policy is associated with.
- policy_name - Name of policy to be refreshed.

Syntax for the ENABLE_POLICY Procedure:

```
DBMS_RLS.ENABLE_POLICY (
   object_schema IN VARCHAR2 := NULL,
   object_name   IN VARCHAR2,
   policy_name   IN VARCHAR2,
   enable        IN BOOLEAN);
```

Where:

- object_schema - Schema containing the table or view (logon user if NULL).
- object_name - Name of table or view that the policy is associated with.
- policy_name - Name of policy to be enabled or disabled.

- Enable - TRUE to enable the policy, FALSE to disable the policy.

Through the use of the above procedures DBAs and developers can easily manage policies.

Summary

Oracle has given DBAs and developers a powerful new tool to manage row level security. This new tool is a combination of contexts, triggers and packages and a new package named DBMS_RLS through which security policies are implemented.

Through the proper use of policies, contexts, packages and database level triggers row level security can be easily integrated in Oracle8i applications.

(The above lesson was excerpted from the soon to be released book: "Oracle8i Administration and Management", Michael R. Ault, John Wiley and Sons Publishers).

Hour 2:

Data Warehouse Loading

The objectives of this section are to:

1. Show the new features of Oracle SQL*Loader that help with data loading
2. Discuss the use of SQL and PL/SQL for data loading
3. Discuss aggregation during loading of data warehouses

Loading Techniques

In Oracle7, Oracle8 and Oracle8i the standard way to load a data warehouse involves multiple steps. The steps involved are usually:

1. Extract data from source database(s)
2. Load data into temporary work tables performing any possible transformation/clean-up and aggregation.
3. Use internal/external scripts and stored objects to transform, aggregate and load the data

The benefits of using temporary tables is that the data is loaded fairly fast into the environment where it can be cleaned, transformed and aggregated using internal procedures which can be a highly automated process.

Alternatively the data can be transformed and aggregated as it is extracted using scripts and procedures and then can be loaded into the database using SQL*Loader. SQL*Loader can be used with a direct load option which prebuilds data blocks and inserts them into the database.

If data resides in non-Oracle data sources, there are gateways which Oracle provides that should be used to make these sources available to your processes:

- Procedural Gateway to APPC – appliations that use IBM's Advanced program-toprogram communication.
- Transparent Gateway for IBM's DB2.

- Transparent Gateway for IBM's DRDA using the DRDA standard APPC/LU 6.2 protocal
- Transparent Gateway for EDA/SQL that allows transparent access to 15 IBM MVS mainframe databases.
- Transparent Gateway to Informix
- Transparent Gateway to RMS on Digital (DEC) (Now Compaq) VAX
- Transparent Gateway to Sybase
- Transparent Gateway to Teradata

Data can be sucked across these gateways into a holding table in an Oracle database for either manipulation or extraction into a flat file depending on how you wish to load the results into the warehouse. Alternatively flat files can be extracted from the data source and transferred to the data warehouse server. If you can have IBM files translated from EBCIDIC format prior to your having to deal with it makes things much easier.

*SQL*Loader direct load options*

SQL*Loader is usually the work horse of the data warehouse loading effort. SQl*Loader can do the following:

- Load data from disk or tape
- Support a large number of data types
- Load into one or more tables based on specified criteria
- Load fixed or variable length records.
- Generate unique derived numeric keys as the data warehouse is loaded
- Support several high performance options
- Produce detailed error reports and reload tables to allow the isolation and correction of erros quickly and easily
- Preprocess data as it is loading it into the database.

If you specify the SQL*Loader command for your platform at the command line without specifying any arguments a usage guide will be printed. Here is an example:

```
SQL*Loader: Release 8.1.5.0.0 - Production on Sun Aug 29 20:10:28 1999

(c) Copyright 1999 Oracle Corporation.  All rights reserved.


Usage: SQLLOAD keyword=value [,keyword=value,...]
Valid Keywords:
    userid -- ORACLE username/password
   control -- Control file name
       log -- Log file name
       bad -- Bad file name
      data -- Data file name
   discard -- Discard file name
discardmax -- Number of discards to allow          (Default all)
      skip -- Number of logical records to skip    (Default 0)
      load -- Number of logical records to load    (Default all)
    errors -- Number of errors to allow            (Default 50)
      rows -- Number of rows in conventional path bind array or between
              direct path data saves (Default: Conventional path 64,
              Direct path all)
  bindsize -- Size of conventional path bind array in bytes
             (Default 65536)
    silent -- Suppress messages during run
              (header,feedback,errors,discards,partitions)
    direct -- use direct path                      (Default FALSE)
   parfile -- parameter file: name of file that contains parameter
              specifications
  parallel -- do parallel load                     (Default FALSE)
      file -- File to allocate extents from
skip_unusable_indexes -- disallow/allow unusable indexes or index
                         partitions  (Default FALSE)
skip_index_maintenance -- do not maintain indexes, mark affected
                          indexes as unusable  (Default FALSE)
commit_discontinued -- commit loaded rows when load is discontinued
                       (Default FALSE)
  readsize -- Size of Read buffer                  (Default 65535)

PLEASE NOTE: Command-line parameters may be specified either by
position or by keywords.  An example of the former case is 'sqlload
scott/tiger foo'; an example of the latter is 'sqlload control=foo
userid=scott/tiger'.  One may specify parameters by position before
but not after parameters specified by keywords.  For example,
'sqlload scott/tiger control=foo logfile=log' is allowed, but
'sqlload scott/tiger control=foo log' is not, even though the
position of the parameter 'log' is correct.
```

One thing to notice when discussing the command line parameters for the SQL*Loader command is the caveat under the ROWS parameter. A direct path load by default commits only after the complete load is completed. What this means for a direct load without a ROWS specification is that you better have enough rollback area to handle the entire load transaction volume. A direct path load is the fastest way to load data into an Oracle database. You specify a direct path load by use of the DIRECT=TRUE keyword pair. By splitting input record sets into multiple files and then using the PARALLEL=TRUE keyword pair you can use multiple SQL*Loader sessions to load data into the same table (each will

get a data segment equal to the setting of the INITIAL parameter to load into, plan for this) allowing almost a direct multiplication of throughput with an equally linear decrease in loading time.

Another key to speeding the load process using SQL*Loader is to eliminate logging as much as possible. You can eliminate all but data dictionary action redo logging by using the NOLOGGING option against the tables to be loaded before beginning the load. To put is simply, issue the

```
ALTER TABLE table-name NOLOGGING;
```

command against the table before beginning the load. Another method is to place the keyword UNRECOVERABLE in the start of the SQL*Loader control file.

A SQL*Loader session will produce a log file and a discard file. The discard and log file will be named the same as the load file with the endings ".bad" and ".log" unless the tool is told otherwise. The arguments can be specified by position or through the use of the keyword with or without an equal sign.

To summarize, in order to minimize the time required to perform a SQL*Load operation you:

1. Set all involved tables to NOLOGGING using ALTER TABLE
2. Split the input data into multiple files
3. Place the UNRECOVERABLE keyword in the control file(s) for the load
4. Use the DIRECT=TRUE and PARALLEL=TRUE keywords

IMPORT-EXPORT

Import and export are two reflexive products, one undoes what the other does. Export extracts a logical copy of the objects and data while Import uses the data from Export to rebuild and reload objects and data. An export file taken from any recent version of Oracle can usually be loaded into any current version at least back one or two main point releases or more.

This logical unload and reload of data performed in EXPORT and IMPORT can be used to move both small and large tables from source Oracle databases to the data warehouse to load into temporary tables for further processing. Generally you will not load the data into the same user as it was extracted from so a FROMUSER-TOUSER IMPORT is generally used. Using the same method as we did with SQL*Loader (except you must specify help=y) here are quick synopsis of the IMP and EXP command sets on an NT version of Oracle:

```
E:\ exp help=y
Export : Release 8.1.5.0 – Production on Sun Aug 29 20:58:21 1999

(c) Copyright 1999 Oracle Corporation. All rights reserved.
```

You can let Export prompt you for parameters by entering the EXP Command followed by your username/password:

```
     Example: EXP SCOTT/TIGER
```

Or, you can control how Export runs by entering the EXP command followed by various arguments. To specify parameters, you use keywords:

```
     Format: EXP KEYWORD=value or KEYWORD=(value2, value2,…,valueN)
     Example: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)
              Or       TABLES=(T1:P1,TI:P2) if on Oracle8 and T1 is
                                  partitioned.
```

USERID must be the first parameter on the command line.

```
Keyword  Description (Default)     Keyword      Description(Default)
USERID   userid/password           FULL         export entire file (N)
BUFFER   size of data buffer       OWNER        list of owner usernames
FILE     output files (EXPDAT.DMP) TABLES       list of table names
COMPRESS import into one extent    RECORDLENGTH length of IO record
GRANTS   export grants (Y)         INCTYPE      incremental export type
INDEXES  export indexes (Y)        RECORD       track incr. Export (Y)
ROWS     export data rows (Y)      PARFILE      parameter filename
CONSTRAINTS export constraints (Y) CONSISTENT   cross-table consistency
LOG      log file of screen output STATISTICS   analyze objects
DIRECT   direct path (N)                       (ESTIMATE)
FEEDBACK display progress every X rows (0)
Oracle8 additions:
POINT_IN_TIME_RECOVER Tablespace point in time recovery (N)
RECOVERY_TABLESPACES List of tablespaces to recover
Oracle8i Additions:
FILESIZE maximum size of each dumpfile
QUERY    select clause used to export a subset of a table

The following keywords only apply to transportable tablespaces (O8i)
TRANSPORT_TABLESPACE export transportable tablespace metadata (N)
TABLESPACES list of tablespaces to transport

Export terminated successfully without warnings

E:\ imp help=y

Import: Release 8.1.5.0.0 – Production on Sun Aug 29 1999 20:58:21 1999

(c) Copyright 1999 Oracle Corporation. All rights reserved.
```

You can let Export prompt you for parameters by entering the EXP Command followed by your username/password:

```
Example: IMP SCOTT/TIGER
```

Or, you can control how Import runs by entering the IMP command followed by various arguments. To specify parameters, you use keywords:

```
Format:  IMP KEYWORD=value or KEYWORD=(value2, value2,…,valueN)
Example: IMP SCOTT/TIGER IGNORE=Y TABLES=(EMP,DEPT) FULL=N
         Or       TABLES=(T1:P1,TI:P2) if on Oracle8 and T1 is
                                 partitioned.
```

USERID must be the first parameter on the command line.

```
Keyword  Description (Default)      Keyword     Description(Default)
USERID   username/password          FULL        import entire file (N)
BUFFER   size of data buffer        FROMUSER    list of owner usernames
FILE     input files (EXPDAT.DMP)   TOUSER      list of usernames
SHOW     just list file contents (N) TABLES     list of table names
IGNORE   ignore create errors (N)   RECORDLENGTH length of IO record
GRANTS   import grants (Y)          INCTYPE     incremental import type
INDEXES  import indexes (Y)         COMMIT      commit array insert (N)
ROWS     import data rows (Y)       PARFILE     parameter filename
LOG      log file of screen output  CONSTRAINTS import constraints (Y)
DESTROY  overwrite tablespace data file (N)
INDEXFILE write table/index info to specified file
SKIP_UNUSABLE_INDEXES skip maintenance of unusable indexes (N)
```

Oracle8 additions:

- CHARSET character set of export file (NLS_LANG)
- POINT_IN_TIME_RECOVER tablespace point-in-time recovery (N)
- ANALYZE execute ANALYZE statments in dump file (Y)
- FEEDBACK display progress every x rows (0)
- Oracle8i additions:
- TOID_VALIDATE skip validation of specified type ids
- FILESIZE maximum size of each dump file
- RECALCULATE_STATISTICS recalculate statistics (N)

The following keywords only apply to transportable tablespaces

- TRANSPORT_TABLESPACE import transportable tablespace metadata (N)
- DATAFILES datafiles to be transported into tablespace
- ITS_OWNERS users that own data in the transportable tablespace set

- Import terminated successfully without warnings.

Both import and export can be run as FULL, OWNER or TABLE level activities. Notice how partitions are specified with the colon separating the table name from the partition name. This use of partition name in the export import utility and the ability to specify it in selects and other command in Oracle8 argue for not letting Oracle do default naming of partitions.

Notice the COMPRESS keyword. The COMPRESS keyword forces all data for the tables in the export to be forced into one extent, the next extent is sized the same as in the original table. If you aren't careful using COMPRESS in export and import you could exceed the size of your import tablespace data files for large tables. Another important IMPORT keyword is IGNORE, if the tables already exist in the target database use the IGNORE keyword to ignore creation errors when objects already exist. Another IMPORT keyword is COMMIT, use COMMIT to specify to commit after the number of rows that will fit into a single buffer are imported. By commiting after a specified number of rows instead of the default which is after a full table is imported you control the size of rollback segment required. If you use commit use an initial buffer specification of around 1,000,000 to start with and move up from there. On some platforms such as SUN Solaris after a certain buffer size is reached no further increases change the number of rows committed.

Data Warehouse Tools

The objectives of this section are to:

1. Provide the student with an overview of Oracle express server
2. Provide the student with an overview of Oracle discoverer
3. Discuss the use of third-party query tools.

An Overview of Oracle Express Server

The Oracle Express system is divided into four main areas, the Oracle Express Server, the Oracle Express Analyzer, the Oracle Express Administrator and the Oracle Express Objects.

Oracle Express Server

Oracle Express Server (OES) is acquired technology. Oracle Corporation bought it from Express IRI. This merged the best database and the best OLAP tool. The OES is a great example of an OLAP toolset. It uses variables, dimensions, formulas and relations to describe a cubic data model that is stored in cached

storage inside the OES domain. This cached storage allows just-in-time collection of data giving good performance.

OES uses time slices of data rather than transaction based data. It helps solve questions like: "How many widgets did we sell for the month of May across all territories?" .

Dimensions index and organize data stored in the variable or calculated by a formula, uniquely identifying each occurrence in the database. Dimensions are usually hierarchical in nature.

A variable roughly approximates a fact table. It is an array that holds data values. Another term that you may hear is cell in relation to variables. A variable can store either numbers or textual information.

Formulas are derived data items that are dynamically calculated. Like a view in standard Oracle a formulas results are never stored, just its definition. A formula is a good way to store complex or frequently used calculations. Formulas can apply to both variables and other formulas.

Relations like the items in one dimension to the items in another dimension. They can be one-to-one or one-to-many. Relations capture hierarchies.

Oracle Express Analyzer

The second component of the OES system is the Oracle Express Analyzer (OEA). The express analyzer allows the building of *briefings*. A briefing is a report that is linked directly to the data, it changes as the data changes.

The OEA uses a standard GUI interface that allows the building and display of briefings. The EA tool consists of a main window which contains a menu bar, a selector toolbox, toolbox, briefing browser, database browser and object inspector. These tools allow building of briefings in a fairly intuitive manner.

Oracle Express Administrator

The third component of the Oracle Express package is the Oracle Express Administrator (also OEA) (confusing isn't it?) is used to create a new database (cache area) which involves defining dimensions, formulas and variables. Once the database is defined the Oracle Express Analyzer is used to create briefings against it.

Oracle Express Objects

The Oracle Express Objects section of the OES is an object-oriented application development tool for creation of graphical OLAP client/server objects.

Other Parts to the Oracle Express Puzzle

There are several more players in the Oracle Express suite of tools, Oracle Financial Analyzer, which allows analysis of data from spreadsheets, outside data sources and Oracle Financials (GL only). Oracle Sales Analyzer, which can access virtually and data source to provide analysis capability and by using the Relational Access manager (RAM) OSA can access virtually any relational database. The final piece is the Oracle Web Agent which allows an OES application to be run on any web browser.

An Overview of Oracle Discoverer

The Discoverer product is an end user query, reporting, drill/pivot and web publishing tool that allows users to gain rapid access to the relational data warehouse allowing them to make more informed business decisions.

The discoverer tool is comprised of three main elements:

- User edition
- Administration edition
- End user layer

User Edition

The user edition enables users to query the warehouse, graph results, create reports, perform drill and pivot analysis and publish results to the World Wide Web. The module is designed for ease of use, performance and flexible warehouse exploration. The tool allows business-oriented users to easily interact directly with the data without having to go to a separate interface to drill down or create data pivots. Usablility is enhanced by Discoverer by the use of wizards, cue cards and CBT modules. A single user interface allows for query, report and drill/down/pivot functionality. Much of the ease of use comes from the similar look and feel between Windows and the Discoverer GUI interfaces.

Administration Edition

The administration edition allows for the administrators to define how the data is presented to the users. The administrator sets up the business areas and folders.

All access is via a GUI interface which is tightly based in the Windows paradigm. Oracle has provided wizards in the administration edition that automate many of the tasks such as creating business areas and folders. The administraiton edition creates the metadata for the end users.

Once a user has created a data report, it can be pushed out to the web as a HTML document.

End User Layer

The end user layer is a server based, low-maintenance, powerful mechanism for providing users with a business-oriented view of the data warehouse. The end user layer abstracts the complexity of the underlying database structures, defines drill down and other related analysis information and automatically creates summary tables.

Discoverer is able to logically relate individual data items, such as the components of an address, into one user item, a total address. This is known as creating a business area. A business area groups data items of interest to a particular group with subsequent level grouped in an hierarchical manner. The business area is grouped into folders and complex folders.

The end user layer is completely defined by the administrator before users are allowed access.

Summary

In this set of lessons on Oracle and Data warehousing we have discuss many features of the various versions of oracle as they relate to data warehousing. The most important feature however has been left out. The most important feature of a well run data warehouse that operates efficiently and quickly is a well trained database administrator. By coming to these seminars, attending Oracle training and participating in OOW, IOUG and online discussion areas you demonstrate your desire to improve your knowledge and grow in Oracle understanding.