Oracle® Database

Data Warehousing Guide

11*g* Release 2 (11.2)

E25554-02

July 2013

ORACLE®

Oracle Database Data Warehousing Guide, 11*g* Release 2 (11.2)

E25554-02

Primary Author: Paul Lane

Contributor: Patrick Amor, Hermann Baer, Mark Bauer, Subhransu Basu, Srikanth Bellamkonda, Randy Bello, Paula Bingham, Tolga Bozkaya, Lucy Burgess, Donna Carver, Rushan Chen, Benoit Dageville, John Haydu, Lilian Hobbs, Hakan Jakobsson, George Lumpkin, Alex Melidis, Valarie Moore, Cetin Ozbutun, Ananth Raghavan, Jack Raitto, Ray Roccaforte, Sankar Subramanian, Gregory Smith, Margaret Taft, Murali Thiyagarajan, Ashish Thusoo, Thomas Tong, Mark Van de Wiel, Jean-Francois Verrier, Gary Vincent, Andreas Walter, Andy Witkowski, Min Xiao, Tsae-Feng Yu

Contents

5 Partitioning in Data Warehouses

6 Parallel Execution in Data Warehouses

7 Indexes

8 Integrity Constraints

9 Basic Materialized Views

10 Advanced Materialized Views

11 Dimensions

Part IV Managing the Data Warehouse Environment

12 Overview of Extraction, Transformation, and Loading

13 Extraction in Data Warehouses

14 Transportation in Data Warehouses

15 Loading and Transformation

16 Maintaining the Data Warehouse

17 Change Data Capture

18 Basic Query Rewrite

19 Advanced Query Rewrite

20 Schema Modeling Techniques

21 SQL for Aggregation in Data Warehouses

22 SQL for Analysis and Reporting

23 SQL for Modeling

Preface

This preface contains these topics:

Audience

This guide is intended for database administrators, system administrators, and database application developers who design, maintain, and use data warehouses.

To use this document, you need to be familiar with relational database concepts, basic Oracle server concepts, and the operating system environment under which you are running Oracle.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit
`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit
`http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Note that this book is meant as a supplement to standard texts about data warehousing. This book focuses on Oracle-specific material and does not reproduce in detail material of a general nature. For additional information, see:

- *The Data Warehouse Toolkit* by Ralph Kimball (John Wiley and Sons, 1996)
- *Building the Data Warehouse* by William Inmon (John Wiley and Sons, 1996)

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in Oracle Database?

This section describes the new features of Oracle Database 11*g* Release 2 (11.2) and provides pointers to additional information. New features information from previous releases is also retained to help those users migrating to the current release.

The following section describes new features in Oracle Database:

- Oracle Database 11g Release 2 (11.2) New Features in Data Warehousing
- Oracle Database 11g Release 1 (11.1) New Features in Data Warehousing

Oracle Database 11g Release 2 (11.2) New Features in Data Warehousing

- Analytic Functions

  New SQL analytic functions have been introduced that enable you to list (or concatenate) measure values within a group (`LISTAGG`). Another new function (`NTH_VALUE`) enables you to retrieve an arbitrary (in other words, nth) record in a window. Finally, the existing functions `LAG` and `LEAD` now have been improved with the addition of the `IGNORE NULLS` option.

  See Also: Chapter 22, "SQL for Analysis and Reporting" for more information

- Preprocessing of Files from External Tables

  You can now specify a program to be executed that will process files and enable Oracle Database to use the output. This preprocessing of files enables you to load large amounts of compressed data without first uncompressing it on a disk.

  See Also: Chapter 15, "Loading and Transformation" for more information

- Materialized View Refresh Enhancements

  Materialized view logs can now be purged outside the refresh process, thus improving performance. An additional performance improvement is with materialized views that contain aggregates, joins, or both. If you use a `WITH COMMIT SCN` clause, materialized view log processing can be optimized, thus speeding up the refresh process.

See Also: Chapter 9, "Basic Materialized Views" and Chapter 16, "Maintaining the Data Warehouse" for more information

Oracle Database 11g Release 1 (11.1) New Features in Data Warehousing

- Pivot and Unpivot Operators

  The `PIVOT` operator makes it easy to create aggregated cross-tabular output that condenses many rows into a compact result set useful for reports. For instance, input data holding sales of one month in each row can be pivoted into output holding twelve months in each row, with each month in its own column. By combining multiple input rows into each output row, `PIVOT` also enables inter-row comparison without a table self-join. The `UNPIVOT` operator reshapes data into a format useful for further relational operations. For example, if a source data set presents twelve months of sales values in each row, `UNPIVOT` can reshape each source row into twelve output rows, each holding one month of sales data. The unpivoted results are in a more normalized relational form than the source data, and they can be manipulated with simpler and more efficient SQL.

  See Also: Chapter 21, "SQL for Aggregation in Data Warehouses" for more information

- Partition Advisor

  The SQL Access Advisor has been enhanced to include partition advice. It recommends the right strategy to partition tables, indexes, and materialized views to get best performance from an application.

  See Also: Chapter 5, "Partitioning in Data Warehouses" for more information

- Change Data Capture (CDC) Enhancements

  CDC is now aware of direct-path load operations and implicit data changes as the result of partition-maintenance operations. Users can now turn synchronous CDC on and off as needed. Also, the flexibility of purging change data from change tables has been improved, so you can specify a date range for which data should be purged.

  Another improvement is that it is easier to maintain a subscription window to change data. You now have control over the definition of the change subscription, so the window can be moved forward and backward.

  See Also: Chapter 17, "Change Data Capture" for more information

- Query Rewrite Enhancements

  Query rewrite has been enhanced to support queries containing inline views. Prior to this release, queries containing inline views could rewrite only if there was an exact text match with the inline views in the materialized views. Because inline views no longer need to textually match between the query and the materialized view, a larger number of queries with inline views can be rewritten. Another significant query rewrite improvement is the ability to rewrite queries that reference remote tables.

See Also: Chapter 18, "Basic Query Rewrite" for more information

- Refresh Enhancements

  Refresh has been enhanced to support automatic index creation for `UNION ALL` materialized views, the use of query rewrite during a materialized view's atomic refresh, and materialized view refresh with set operators. Also, partition change tracking refresh of `UNION ALL` materialized views is now possible. Finally, catalog views have been enhanced to contain information on the staleness of partitioned materialized views. These improvements will lead to faster refresh performance.

  See Also: Chapter 16, "Maintaining the Data Warehouse" for more information

- Oracle OLAP Option Data Warehousing Features

  The OLAP Option of the Oracle Database has been enhanced with several features designed to make OLAP cubes attractive alternatives to tables for managing and querying aggregate data in the data warehouse. These include:

  – Further integration of cubes into the SQL query engine. Advancements include integration of cubes with the Oracle query optimizer and a cube row source. These features dramatically increase the efficiency of SQL queries that select from OLAP cubes and dimensions by pushing joins directly into the Oracle Database's multidimensional engine, allowing efficient joins between tables and cubes and by improving overall row/second throughput when selecting from cubes.

  – Automatic query rewrite to cube organized materialized views. Cube-organized materialized views access data from OLAP cubes rather than tables. Like table-based materialized views, application can write queries to detail tables or views and let the database automatically rewrite the query to pre-aggregated data in the cube.

  – Database-managed automatic refresh of cubes. In this release, cubes can be refreshed using the `DBMS_MVIEW.REFRESH` program, just like table-based materialized views. Cubes provide excellent support for `FAST` (incremental) refresh.

  – Cost-based aggregation. In many situations, cubes are much more efficient at managing aggregate data as compared to tables. Cost-based aggregation improves upon these advantages by improving the efficiency of pre-aggregating and querying aggregate data, and by simplifying the process of managing aggregate data.

  Database administrators who support dimensionally modeled data sets (for example, star/snowflake schema) for query by business intelligence tools and applications should consider using OLAP cubes as a summary management solution because they may offer significant performance advantages.

Part I

Concepts

This section introduces basic data warehousing concepts.

It contains the following chapter:

- Chapter 1, "Data Warehousing Concepts"

1

Data Warehousing Concepts

This chapter provides an overview of the Oracle data warehousing implementation. It includes:

- What is a Data Warehouse?
- Data Warehouse Architectures
- Extracting Information from a Data Warehouse

Note that this book is meant as a supplement to standard texts about data warehousing. This book focuses on Oracle-specific material and does not reproduce in detail material of a general nature. Two standard texts are:

- *The Data Warehouse Toolkit* by Ralph Kimball (John Wiley and Sons, 1996)
- *Building the Data Warehouse* by William Inmon (John Wiley and Sons, 1996)

What is a Data Warehouse?

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources. This helps in:

- Maintaining historical records
- Analyzing the data to gain a better understanding of the business and to improve the business.

In addition to a relational database, a data warehouse environment can include an extraction, transportation, transformation, and loading (ETL) solution, statistical analysis, reporting, data mining capabilities, client analysis tools, and other applications that manage the process of gathering data, transforming it into useful, actionable information, and delivering it to business users.

> **See Also:** Chapter 12, "Overview of Extraction, Transformation, and Loading"

A common way of introducing data warehousing is to refer to the characteristics of a data warehouse as set forth by William Inmon:

- Subject Oriented
- Integrated
- Nonvolatile

- Time Variant

Subject Oriented

Data warehouses are designed to help you analyze data. For example, to learn more about your company's sales data, you can build a data warehouse that concentrates on sales. Using this data warehouse, you can answer questions such as "Who was our best customer for this item last year?" or "Who is likely to be our best customer next year?" This ability to define a data warehouse by subject matter, sales in this case, makes the data warehouse subject oriented.

Integrated

Integration is closely related to subject orientation. Data warehouses must put data from disparate sources into a consistent format. They must resolve such problems as naming conflicts and inconsistencies among units of measure. When they achieve this, they are said to be integrated.

Nonvolatile

Nonvolatile means that, once entered into the data warehouse, data should not change. This is logical because the purpose of a data warehouse is to enable you to analyze what has occurred.

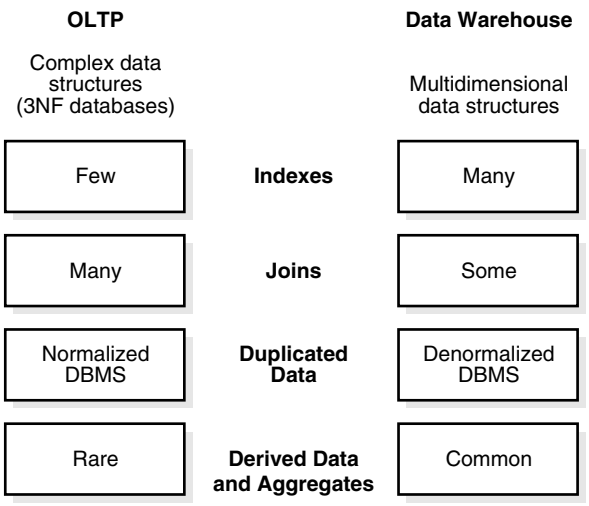
Time Variant

A data warehouse's focus on change over time is what is meant by the term time variant. In order to discover trends and identify hidden patterns and relationships in business, analysts need large amounts of data. This is very much in contrast to **online transaction processing (OLTP)** systems, where performance requirements demand that historical data be moved to an archive.

Contrasting OLTP and Data Warehousing Environments

Figure 1–1 illustrates key differences between an OLTP system and a data warehouse.

Figure 1–1 Contrasting OLTP and Data Warehousing Environments



One major difference between the types of system is that data warehouses are not usually in **third normal form (3NF)**, a type of data normalization common in OLTP environments.

Data warehouses and OLTP systems have very different requirements. Here are some examples of differences between typical data warehouses and OLTP systems:

- Workload

  Data warehouses are designed to accommodate *ad hoc* queries and data analysis. You might not know the workload of your data warehouse in advance, so a data warehouse should be optimized to perform well for a wide variety of possible query and analytical operations.

  OLTP systems support only predefined operations. Your applications might be specifically tuned or designed to support only these operations.

- Data modifications

  A data warehouse is updated on a regular basis by the ETL process (run nightly or weekly) using bulk data modification techniques. The end users of a data warehouse do not directly update the data warehouse except when using analytical tools, such as data mining, to make predictions with associated probabilities, assign customers to market segments, and develop customer profiles.

  In OLTP systems, end users routinely issue individual data modification statements to the database. The OLTP database is always up to date, and reflects the current state of each business transaction.

- Schema design

  Data warehouses often use denormalized or partially denormalized schemas (such as a star schema) to optimize query and analytical performance.

  OLTP systems often use fully normalized schemas to optimize update/insert/delete performance, and to guarantee data consistency.

- Typical operations

  A typical data warehouse query scans thousands or millions of rows. For example, "Find the total sales for all customers last month."

  A typical OLTP operation accesses only a handful of records. For example, "Retrieve the current order for this customer."

- Historical data

  Data warehouses usually store many months or years of data. This is to support historical analysis and reporting.

  OLTP systems usually store data from only a few weeks or months. The OLTP system stores only historical data as needed to successfully meet the requirements of the current transaction.

Data Warehouse Architectures

Data warehouses and their architectures vary depending upon the specifics of an organization's situation. Three common architectures are:

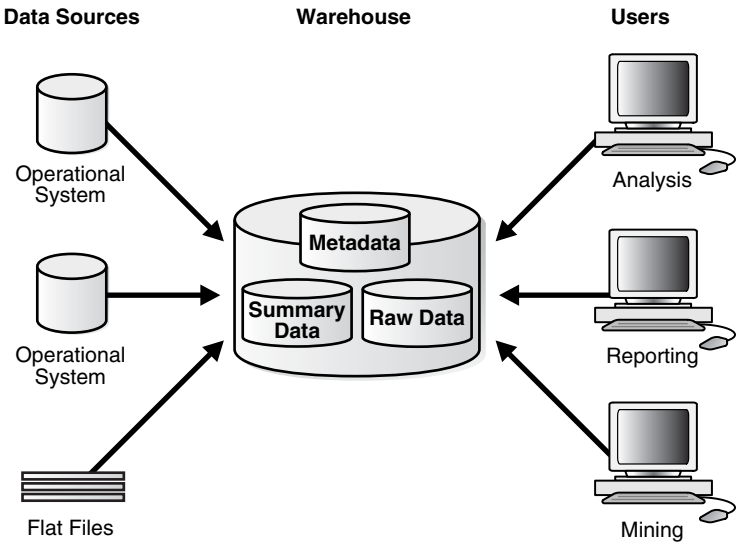
- Data Warehouse Architecture: Basic
- Data Warehouse Architecture: with a Staging Area

- Data Warehouse Architecture: with a Staging Area and Data Marts

Data Warehouse Architecture: Basic

Figure 1–2 shows a simple architecture for a data warehouse. End users directly access data derived from several source systems through the data warehouse.

Figure 1–2 Architecture of a Data Warehouse

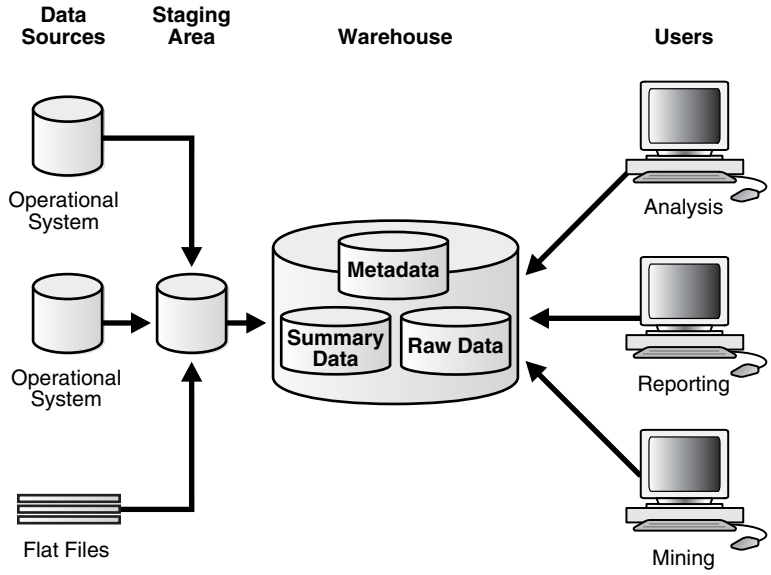


In Figure 1–2, the metadata and raw data of a traditional OLTP system is present, as is an additional type of data, summary data. Summaries are very valuable in data warehouses because they pre-compute long operations in advance. For example, a typical data warehouse query is to retrieve something such as August sales. A summary in an Oracle database is called a **materialized view**.

Data Warehouse Architecture: with a Staging Area

You must clean and process your operational data before putting it into the warehouse, as shown in Figure 1–3. You can do this programmatically, although most data warehouses use a **staging area** instead. A staging area simplifies building summaries and general warehouse management. Figure 1–3 illustrates this typical architecture.

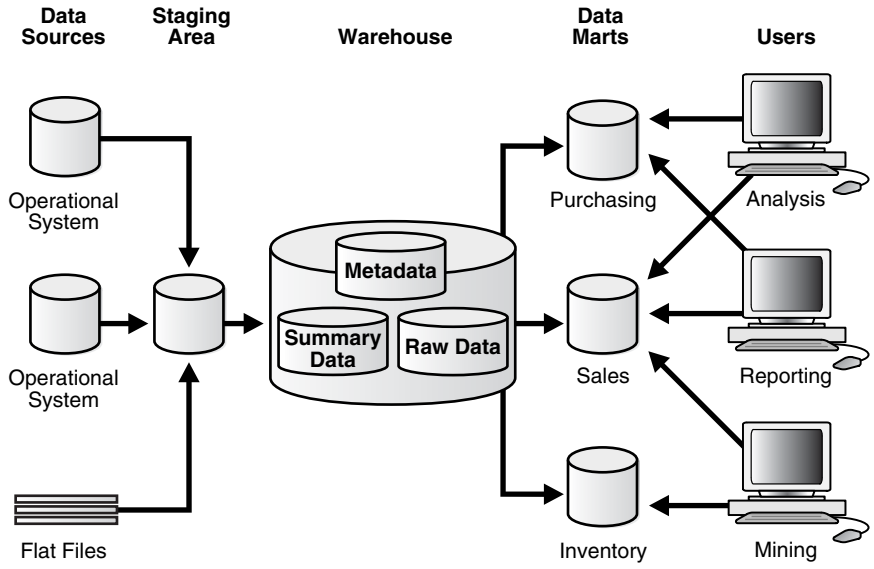
Figure 1–3 Architecture of a Data Warehouse with a Staging Area



Data Warehouse Architecture: with a Staging Area and Data Marts

Although the architecture in Figure 1–3 is quite common, you may want to customize your warehouse's architecture for different groups within your organization. You can do this by adding **data marts**, which are systems designed for a particular line of business. Figure 1–4 illustrates an example where purchasing, sales, and inventories are separated. In this example, a financial analyst might want to analyze historical data for purchases and sales or mine historical data to make predictions about customer behavior.

Figure 1–4 Architecture of a Data Warehouse with a Staging Area and Data Marts



Note: Data marts are an important part of many data warehouses, but they are not the focus of this book.

Extracting Information from a Data Warehouse

You can extract information from the masses of data stored in a data warehouse by analyzing the data. The Oracle Database provides several ways to analyze data:

- A wide array of statistical functions, including descriptive statistics, hypothesis testing, correlations analysis, test for distribution fit, cross tabs with Chi-square statistics, and analysis of variance (ANOVA); these functions are described in the *Oracle Database SQL Language Reference*.
- OLAP
- Data Mining

OLAP

Oracle Database offers the industry's first and only embedded OLAP server. Oracle OLAP provides native multidimensional storage and speed-of-thought response times when analyzing data across multiple dimensions. The database provides rich support for analytics such as time series calculations, forecasting, advanced aggregation with additive and non additive operators, and allocation operators. These capabilities make the Oracle database a complete analytical platform, capable of supporting the entire spectrum of business intelligence and advanced analytical applications.

Oracle OLAP uses a multidimensional data model to perform complex statistical, mathematical, and financial analysis of historical data in real time. Oracle OLAP is fully integrated in the database, so that you can use standard SQL administrative, querying, and reporting tools.

For more information regarding OLAP, see *Oracle OLAP User's Guide*.

Full Integration of Multidimensional Technology

By integrating multidimensional objects and analytics into the database, Oracle provides the best of both worlds: the power of multidimensional analysis along with the reliability, availability, security, and scalability of the Oracle database.

Oracle OLAP is fully integrated into Oracle Database. At a technical level, this means:

- The OLAP engine runs within the kernel of Oracle Database.
- Dimensional objects are stored in Oracle Database in their native multidimensional format.
- Cubes and other dimensional objects are first class data objects represented in the Oracle data dictionary.
- Data security is administered in the standard way, by granting and revoking privileges to Oracle Database users and roles.
- Applications can query dimensional objects using SQL.

The benefits to your organization are significant. Oracle OLAP offers the power of simplicity. One database, standard administration and security, standard interfaces and development tools.

Ease of Application Development

Oracle OLAP makes it easy to enrich your database and your applications with interesting analytic content. Native SQL access to Oracle multidimensional objects and calculations greatly eases the task of developing dashboards, reports, business intelligence (BI) and analytical applications of any type compared to systems that offer

proprietary interfaces. Moreover, SQL access means that the power of Oracle OLAP analytics can be used by *any* database application, not just by the traditional limited collection of OLAP applications.

Ease of Administration

Because Oracle OLAP is completely embedded in the Oracle database, there is no administration learning curve as is typically associated with standalone OLAP servers. You can leverage your existing DBA staff, rather than invest in specialized administration skills.

One major administrative advantage of Oracle's embedded OLAP technology is automated cube maintenance. With standalone OLAP servers, the burden of refreshing the cube is left entirely to the administrator. This can be a complex and potentially error-prone job. The administrator must create procedures to extract the changed data from the relational source, move the data from the source system to the system running the standalone OLAP server, load and rebuild the cube. The DBA must take responsibility for the security of the deltas (changed values) during this process as well.

With Oracle OLAP, in contrast, cube refresh is handled entirely by the Oracle database. The database tracks the staleness of the dimensional objects, automatically keeps track of the deltas in the source tables, and automatically applies only the changed values during the refresh process. The DBA simply schedules the refresh at appropriate intervals, and Oracle Database takes care of everything else.

Security

With Oracle OLAP, standard Oracle Database security features are used to secure your multidimensional data.

In contrast, with a standalone OLAP server, administrators must manage security twice: once on the relational source system and again on the OLAP server system. Additionally, they must manage the security of data in transit from the relational system to the standalone OLAP system.

Unmatched Performance and Scalability

Business intelligence and analytical applications are dominated by actions such as drilling up and down hierarchies and comparing aggregate values such as period-over-period, share of parent, projections onto future time periods, and a myriad of similar calculations. Often these actions are essentially random across the entire space of potential hierarchical aggregations. Because Oracle OLAP pre-computes or efficiently computes on the fly all aggregates in the defined multidimensional space, it delivers unmatched performance for typical business intelligence applications.

Oracle OLAP queries take advantage of Oracle shared cursors, dramatically reducing memory requirements and increasing performance.

When Oracle Database is installed with Real Application Clusters (Oracle RAC), OLAP applications receive the same benefits in performance, scalability, fail over, and load balancing as any other application.

Reduced Costs

All these features add up to reduced costs. Administrative costs are reduced because existing personnel skills can be leveraged. Moreover, the Oracle database can manage the refresh of dimensional objects, a complex task left to administrators in other systems. Standard security reduces administration costs as well. Application

development costs are reduced because the availability of a large pool of application developers who are SQL knowledgeable, and a large collection of SQL-based development tools means applications can be developed and deployed more quickly. Any SQL-based development tool can take advantage of Oracle OLAP. Hardware costs are reduced by Oracle OLAP's efficient management of aggregations, use of shared cursors, and Oracle RAC, which enables highly scalable systems to be built from low-cost commodity components.

Querying Dimensional Objects

Oracle OLAP adds power to your SQL applications by providing extensive analytic content and fast query response times. A SQL query interface enables any application to query cubes and dimensions without any knowledge of OLAP.

The OLAP option automatically generates a set of relational views on cubes, dimensions, and hierarchies. SQL applications query these views to display the information-rich contents of these objects to analysts and decision makers. You can also create custom views that comply with the structure expected by your applications, using the system-generated views like base tables.

Analysts can choose any SQL query and analysis tool for selecting, viewing, and analyzing the data You can use your favorite tool or application, or use one of the tools supplied with Oracle Database, such as Oracle Application Express and Business Intelligence Publisher.

Efficient Storage and Uniform Availability of Summary Data

Cube materialized views bring the fast update and fast query capabilities of the OLAP option to applications that query detail relational tables, as well as to applications that query cubes directly.

A single cube materialized view can replace many of the relational materialized views of summaries on a fact table, providing uniform response time to all summary data through query rewrite. Applications experience excellent query performance.

Cube materialized views are cubes that have been enhanced to use the automatic refresh and query rewrite features of Oracle Database. Summary data is generated and stored in a cube, and query rewrite automatically redirects queries to the cube materialized views.

Many of the same data dictionary views and PL/SQL packages that support relational materialized views also support cube materialized views. Moreover, a group of PL/SQL subprograms in `DBMS_CUBE` supports the rapid deployment of cube materialized views from existing relational materialized views.

Tools for Creating and Managing Dimensional Objects

Analytic Workspace Manager is the primary tool for creating, developing, and managing dimensional objects in Oracle Database.

Oracle OLAP is contained in the database and its resources are managed using the same tools, such as Oracle Enterprise Manager Database Control, Automatic Workload Repository, and Automatic Database Diagnostic Monitor.

Data Mining

Data mining uses large quantities of data to create models. These models can provide insights that are revealing, significant, and valuable. For example, you can use data mining to:

- Predict those customers likely to change service providers.
- Discover the factors involved with a disease.
- Identify fraudulent behavior.

Data mining solves many kinds of business problems. For example, data mining can be used to predict customers likely to attrite.

Oracle Data Mining performs data mining in the Oracle Database. Oracle Data Mining does not require data movement between the database and an external mining server, thereby eliminating redundancy, improving efficient data storage and processing, ensuring that up-to-date data is used, and maintaining data security.

For detailed information about Oracle Data Mining, see *Oracle Data Mining Concepts*.

Oracle Data Mining Functionality

Oracle Data Mining supports the major data mining functions. There is at least one algorithm for each data mining function.

Oracle Data Mining supports the following data mining functions:

- Classification: Grouping items into discrete classes and predicting which class an item belongs to; classification algorithms are Decision Tree, Naive Bayes, Generalized Linear Models (Binary Logistic Regression), and Support Vector Machines.
- Regression: Approximating and predicting continuous numeric values; the algorithms for regression are Support Vector Machines and Generalized Linear Models (Multivariate Linear Regression).
- Anomaly Detection: Detecting anomalous cases, such as fraud and intrusions; the algorithm for anomaly detection is one-class Support Vector Machines.
- Attribute Importance: Identifying the attributes that have the strongest relationships with the target attribute (for example, customers likely to churn); the algorithm for attribute importance is Minimum Descriptor Length.
- Clustering: Finding natural groupings in the data that are often used for identifying customer segments; the algorithms for clustering are *k*-Means and O-Cluster.
- Associations: Analyzing "market baskets", items that are likely to be purchased together; the algorithm for associations is a priori.
- Feature Extraction: Creating new attributes (features) as a combination of the original attributes; the algorithm for feature extraction is Non-Negative Matrix Factorization.

In addition to mining structured data, Oracle Data Mining permits mining of text data (such as police reports, customer comments, or physician's notes) or spatial data.

Oracle Data Mining Interfaces

Oracle Data Mining APIs provide extensive support for building applications that automate the extraction and dissemination of data mining insights.

Data mining activities such as model building, testing, and scoring are accomplished through a PL/SQL API, a Java API, and SQL Data Mining functions. The Java API is compliant with the data mining standard JSR 73. The Java API and the PL/SQL API are fully interoperable.

Oracle Data Mining allows the creation of a supermodel, that is, a model that contains the instructions for its own data preparation. The embedded data preparation can be implemented automatically and/or manually. Embedded Data Preparation supports user-specified data transformations; Automatic Data Preparation supports algorithm-required data preparation, such as binning, normalization, and outlier treatment.

SQL Data Mining functions support the scoring of classification, regression, clustering, and feature extraction models. Within the context of standard SQL statements, pre-created models can be applied to new data and the results returned for further processing, just like any other SQL query.

Predictive Analytics automates the process of data mining. Without user intervention, Predictive Analytics routines manage data preparation, algorithm selection, model building, and model scoring so that the user can benefit from data mining without having to be a data mining expert.

Oracle Data Miner is the graphical user interface for Oracle Data Mining. Oracle Data Miner guides you through the data preparation, data mining, model evaluation, and model scoring process. For more information about the Oracle Data Mining interfaces, see *Oracle Data Mining Concepts*.

Part II

Logical Design

This section deals with the issues in logical design in a data warehouse.

It contains the following chapter:

- Chapter 2, "Logical Design in Data Warehouses"

2

Logical Design in Data Warehouses

This chapter explains how to create a logical design for a data warehousing environment and includes the following topics:

Logical Versus Physical Design in Data Warehouses

Your organization has decided to build a data warehouse. You have defined the business requirements and agreed upon the scope of your application, and created a conceptual design. Now you need to translate your requirements into a system deliverable. To do so, you create the logical and physical design for the data warehouse. You then define:

- The specific data content
- Relationships within and between groups of data
- The system environment supporting your data warehouse
- The data transformations required
- The frequency with which data is refreshed

The logical design is more conceptual and abstract than the physical design. In the logical design, you look at the logical relationships among the objects. In the physical design, you look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup/recovery perspective.

Orient your design toward the needs of the end users. End users typically want to perform analysis and look at aggregated data, rather than at individual transactions. However, end users might not know what they need until they see it. In addition, a well-planned design allows for growth and changes as the needs of users change and evolve.

By beginning with the logical design, you focus on the information requirements and save the implementation details for later.

Creating a Logical Design

A logical design is conceptual and abstract. You do not deal with the physical implementation details yet. You deal only with defining the types of information that you need.

One technique you can use to model your organization's logical information requirements is entity-relationship modeling. Entity-relationship modeling involves identifying the things of importance (entities), the properties of these things (attributes), and how they are related to one another (relationships).

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An **entity** represents a chunk of information. In relational databases, an entity often maps to a table. An **attribute** is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column.

To ensure that your data is consistent, you must use unique identifiers. A **unique identifier** is something you add to tables so that you can differentiate between the same item when it appears in different places. In a physical design, this is usually a primary key.

While entity-relationship diagramming has traditionally been associated with highly normalized models such as OLTP applications, the technique is still useful for data warehouse design in the form of dimensional modeling. In dimensional modeling, instead of seeking to discover atomic units of information (such as entities and attributes) and all of the relationships between them, you identify which information belongs to a central fact table and which information belongs to its associated dimension tables. You identify business subjects or fields of data, define relationships between business subjects, and name the attributes for each subject.

> **See Also:** Chapter 11, "Dimensions" for further information regarding dimensions

Your logical design should result in (1) a set of entities and attributes corresponding to fact tables and dimension tables and (2) a model of operational data from your source into subject-oriented information in your target data warehouse schema.

You can create the logical design using a pen and paper, or you can use a design tool such as Oracle Warehouse Builder (specifically designed to support modeling the ETL process).

> **See Also:** Oracle Warehouse Builder documentation set

Data Warehousing Schemas

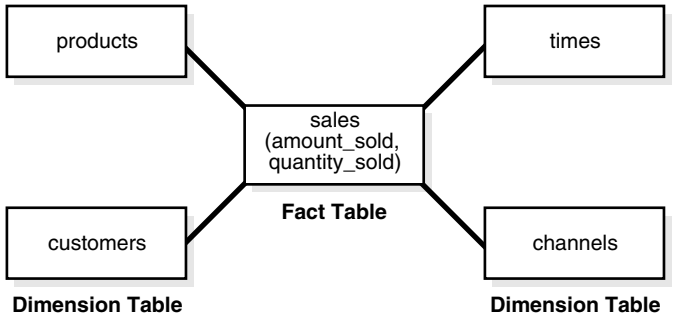
A schema is a collection of database objects, including tables, views, indexes, and synonyms. You can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model.

The model of your source data and the requirements of your users help you design the data warehouse schema. You can sometimes get the source model from your company's enterprise data model and reverse-engineer the logical data model for the data warehouse from this. The physical implementation of the logical data warehouse model may require some changes to adapt it to your system parameters—size of computer, number of users, storage capacity, type of network, and software.

Star Schemas

The star schema is the simplest data warehouse schema. It is called a star schema because the diagram resembles a star, with points radiating from a center. The center of the star consists of one or more fact tables and the points of the star are the dimension tables, as shown in Figure 2–1.

Figure 2–1 Star Schema



The most natural way to model a data warehouse is as a star schema, where only one join establishes the relationship between the fact table and any one of the dimension tables.

A star schema optimizes performance by keeping queries simple and providing fast response time. All the information about each level is stored in one row.

Other Data Warehousing Schemas

Some schemas in data warehousing environments use third normal form rather than star schemas. Another schema that is sometimes useful is the snowflake schema, which is a star schema with normalized dimensions in a tree structure. Another alternative is provided by OLAP, which supports dimensional data types such as cubes and dimensions within Oracle Database.

> **See Also:** Chapter 20, "Schema Modeling Techniques" for further information regarding star and snowflake schemas in data warehouses, *Oracle Database Concepts* for further conceptual material, *Oracle OLAP User's Guide* for more information regarding OLAP schemas

Data Warehousing Objects

Fact tables and dimension tables are the two types of objects commonly used in dimensional data warehouse schemas.

Fact tables are the large tables in your data warehouse schema that store business measurements. Fact tables typically contain facts and foreign keys to the dimension tables. Fact tables represent data, usually numeric and additive, that can be analyzed and examined. Examples include `sales`, `cost`, and `profit`.

Dimension tables, also known as lookup or reference tables, contain the relatively static data in the data warehouse. Dimension tables store the information you normally use to contain queries. Dimension tables are usually textual and descriptive and you can use them as the row headers of the result set. Examples are `customers` or `products`.

Data Warehousing Objects: Fact Tables

A fact table typically has two types of columns: those that contain numeric facts (often called measurements), and those that are foreign keys to dimension tables. A fact table contains either detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi-additive or non-additive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Non-additive facts cannot be added at all. An example of this is averages. Semi-additive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels, where you cannot tell what a level means simply by looking at it.

Requirements of Fact Tables

You must define a fact table for each star schema. From a modeling standpoint, the primary key of the fact table is usually a composite key that is made up of all of its foreign keys.

Data Warehousing Objects: Dimension Tables

A dimension is a structure, often composed of one or more hierarchies, that categorizes data. Dimensional attributes help to describe the dimensional value. They are normally descriptive, textual values. Several distinct dimensions, combined with facts, enable you to answer business questions. Commonly used dimensions are customers, products, and time.

Dimension data is typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysis. These natural rollups or aggregations within a dimension table are called hierarchies.

Hierarchies

Hierarchies are logical structures that use ordered levels to organize data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level. A hierarchy can also be used to define a navigational drill path and to establish a family structure.

Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels. A dimension can be composed of more than one hierarchy. For example, in the product dimension, there might be two hierarchies—one for product categories and one for product suppliers.

Dimension hierarchies also group levels from general to granular. Query tools use hierarchies to enable you to drill down into your data to view different levels of granularity. This is one of the key benefits of a data warehouse.

When designing hierarchies, you must consider the relationships in business structures. For example, a divisional multilevel sales organization.

Hierarchies impose a family structure on dimension values. For a particular level value, a value at the next higher level is its parent, and values at the next lower level are its children. These familial relationships enable analysts to access data quickly.

Levels A level represents a position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the month, quarter, and year levels.

Levels range from general to specific, with the root level as the highest or most general level. The levels in a dimension are organized into one or more hierarchies.

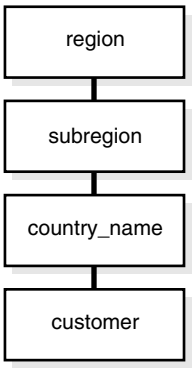
Level Relationships Level relationships specify top-to-bottom ordering of levels from most general (the root) to most specific information. They define the parent-child relationship between the levels in a hierarchy.

Hierarchies are also essential components in enabling more complex rewrites. For example, the database can aggregate an existing sales revenue on a quarterly base to a yearly aggregation when the dimensional dependencies between quarter and year are known.

Typical Dimension Hierarchy

Figure 2–2 illustrates a dimension hierarchy based on `customers`.

Figure 2–2 Typical Levels in a Dimension Hierarchy



See Also: Chapter 11, "Dimensions" and Chapter 18, "Basic Query Rewrite" for further information regarding hierarchies

Data Warehousing Objects: Unique Identifiers

Unique identifiers are specified for one distinct record in a dimension table. Artificial unique identifiers are often used to avoid the potential problem of unique identifiers changing. Unique identifiers are represented with the # character. For example, `#customer_id`.

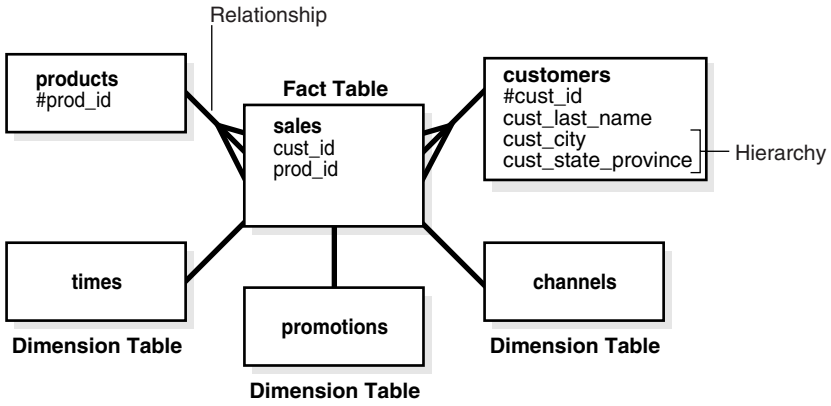
Data Warehousing Objects: Relationships

Relationships guarantee business integrity. An example is that if a business sells something, there is obviously a customer and a product. Designing a relationship between the sales information in the fact table and the dimension tables products and customers enforces the business rules in databases.

Example of Data Warehousing Objects and Their Relationships

Figure 2–3 illustrates a common example of a `sales` fact table and dimension tables `customers`, `products`, `promotions`, `times`, and `channels`.

Figure 2–3 Typical Data Warehousing Objects



Part III

Physical Design

This section deals with the physical design of a data warehouse.

It contains the following chapters:

3

Physical Design in Data Warehouses

This chapter describes the physical design of a data warehousing environment, and includes the following topics:

Moving from Logical to Physical Design

Logical design is what you draw with a pen and paper or design with Oracle Warehouse Builder or Oracle Designer before building your data warehouse. Physical design is the creation of the database with SQL statements.

During the physical design process, you convert the data gathered during the logical design phase into a description of the physical database structure. Physical design decisions are mainly driven by query performance and database maintenance aspects. For example, choosing a partitioning strategy that meets common query requirements enables Oracle Database to take advantage of partition pruning, a way of narrowing a search before performing it.

See Also:

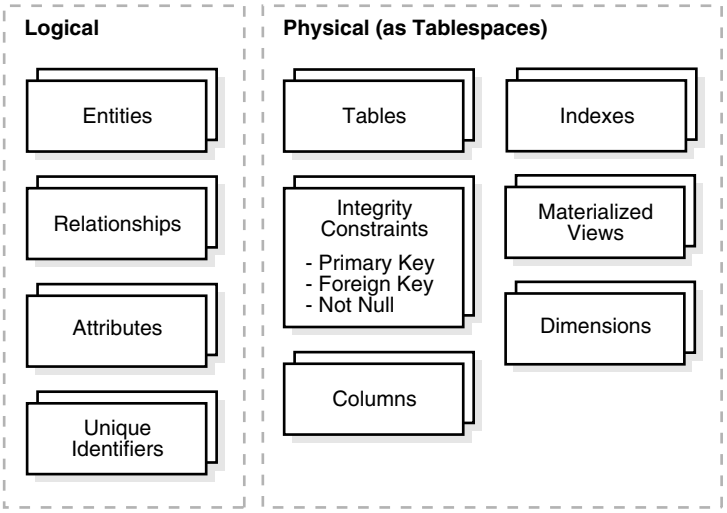
- Chapter 5, "Partitioning in Data Warehouses" for further information regarding partitioning
- *Oracle Database Concepts* for further conceptual material regarding all design matters

Physical Design

During the logical design phase, you defined a model for your data warehouse consisting of entities, attributes, and relationships. The entities are linked together using relationships. Attributes are used to describe the entities. The **unique identifier** (UID) distinguishes between one instance of an entity and another.

Figure 3–1 illustrates a graphical way of distinguishing between logical and physical designs.

Figure 3–1 Logical Design Compared with Physical Design



During the physical design process, you translate the expected schemas into actual database structures. At this time, you must map:

- Entities to tables
- Relationships to foreign key constraints
- Attributes to columns
- Primary unique identifiers to primary key constraints
- Unique identifiers to unique key constraints

Physical Design Structures

Once you have converted your logical design to a physical one, you must create some or all of the following structures:

- Tablespaces
- Tables and Partitioned Tables
- Views
- Integrity Constraints
- Dimensions

Some of these structures require disk space. Others exist only in the data dictionary. Additionally, the following structures may be created for performance improvement:

- Indexes and Partitioned Indexes
- Materialized Views

Tablespaces

A tablespace consists of one or more datafiles, which are physical structures within the operating system you are using. A datafile is associated with only one tablespace. From a design perspective, tablespaces are containers for physical design structures.

Tablespaces need to be separated by differences. For example, tables should be separated from their indexes and small tables should be separated from large tables.

Tablespaces should also represent logical business units if possible. Because a tablespace is the coarsest granularity for backup and recovery or the transportable tablespaces mechanism, the logical business design affects availability and maintenance operations.

You can now use ultralarge data files, a significant improvement in very large databases.

> **See Also:** Chapter 4, "Hardware and I/O Considerations in Data Warehouses" for information regarding tablespaces

Tables and Partitioned Tables

Tables are the basic unit of data storage. They are the container for the expected amount of raw data in your data warehouse.

Using partitioned tables instead of nonpartitioned ones addresses the key problem of supporting very large data volumes by allowing you to divide them into smaller and more manageable pieces. The main design criterion for partitioning is manageability, though you also see performance benefits in most cases because of partition pruning or intelligent parallel processing. For example, you might choose a partitioning strategy based on a sales transaction date and a monthly granularity. If you have four years' worth of data, you can delete a month's data as it becomes older than four years with a single, fast DDL statement and load new data while only affecting 1/48th of the complete table. Business questions regarding the last quarter only affect three months, which is equivalent to three partitions, or 3/48ths of the total volume.

Partitioning large tables improves performance because each partitioned piece is more manageable. Typically, you partition based on transaction dates in a data warehouse. For example, each month, one month's worth of data can be assigned its own partition.

Table Compression

You can save disk space, increase memory efficiency, and improve query performance by compressing heap-organized tables. This often leads to better scalability and query performance. You can enable compression at the tablespace, table, or partition level. A typical type of heap-organized table you should consider for table compression is partitioned tables. Although compressed tables or partitions are updatable, there is some overhead in updating these tables, and high update activity may work against compression by causing some space to be wasted.

OLTP table compression is best suited for tables with significant update activity. Hybrid Columnar Compression, a feature of certain Oracle storage systems, utilizes a combination of both row and columnar methods for storing data. When data is loaded, groups of rows are stored in columnar format, with the values for a given column stored and compressed together. Storing column data together, with the same data type and similar characteristics, drastically increases the storage savings achieved from compression. Hybrid Columnar Compression provides multiple levels of compression and is best suited for tables or partitions with minimal update activity.

> **See Also:**
>
> - *Oracle Database VLDB and Partitioning Guide*
> - Chapter 16, "Maintaining the Data Warehouse"
> - *Oracle Database Administrator's Guide*
> - *Oracle Database Concepts* for more information about Hybrid Columnar Compression

Views

A view is a tailored presentation of the data contained in one or more tables or other views. A view takes the output of a query and treats it as a table. Views do not require any space in the database.

See Also: *Oracle Database Concepts*

Integrity Constraints

Integrity constraints are used to enforce business rules associated with your database and to prevent having invalid information in the tables. Integrity constraints in data warehousing differ from constraints in OLTP environments. In OLTP environments, they primarily prevent the insertion of invalid data into a record, which is not a big problem in data warehousing environments because accuracy has already been guaranteed. In data warehousing environments, constraints are only used for query rewrite. `NOT NULL` constraints are particularly common in data warehouses. Under some specific circumstances, constraints need space in the database. These constraints are in the form of the underlying unique index.

See Also: Chapter 8, "Integrity Constraints"

Indexes and Partitioned Indexes

Indexes are optional structures associated with tables or clusters. In addition to the classical B-tree indexes, bitmap indexes are very common in data warehousing environments. Bitmap indexes are optimized index structures for set-oriented operations. Additionally, they are necessary for some optimized data access methods such as star transformations.

Indexes are just like tables in that you can partition them, although the partitioning strategy is not dependent upon the table structure. Partitioning indexes makes it easier to manage the data warehouse during refresh and improves query performance.

See Also: Chapter 7, "Indexes" and Chapter 16, "Maintaining the Data Warehouse"

Materialized Views

Materialized views are query results that have been stored in advance so long-running calculations are not necessary when you actually execute your SQL statements. From a physical design point of view, materialized views resemble tables or partitioned tables and behave like indexes in that they are used transparently and improve performance.

See Also: Chapter 9, "Basic Materialized Views"

Dimensions

A dimension is a schema object that defines hierarchical relationships between columns or column sets. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next one. A dimension is a container of logical relationships and does not require any space in the database. A typical dimension is city, state (or province), region, and country.

See Also: Chapter 11, "Dimensions"

4

Hardware and I/O Considerations in Data Warehouses

This chapter explains some of the hardware and I/O issues in a data warehousing environment and includes the following topics:

- Overview of Hardware and I/O Considerations in Data Warehouses
- Storage Management

Overview of Hardware and I/O Considerations in Data Warehouses

I/O performance should always be a key consideration for data warehouse designers and administrators. The typical workload in a data warehouse is especially I/O intensive, with operations such as large data loads and index builds, creation of materialized views, and queries over large volumes of data. The underlying I/O system for a data warehouse should be designed to meet these heavy requirements.

In fact, one of the leading causes of performance issues in a data warehouse is poor I/O configuration. Database administrators who have previously managed other systems will likely need to pay more careful attention to the I/O configuration for a data warehouse than they may have previously done for other environments.

This chapter provides the following five high-level guidelines for data-warehouse I/O configurations:

- Configure I/O for Bandwidth not Capacity
- Stripe Far and Wide
- Use Redundancy
- Test the I/O System Before Building the Database
- Plan for Growth

The I/O configuration used by a data warehouse depends on the characteristics of the specific storage and server capabilities, so the material in this chapter is only intended to provide guidelines for designing and tuning an I/O system.

> **See Also:** *Oracle Database Performance Tuning Guide* for additional information on I/O configurations and tuning

Configure I/O for Bandwidth not Capacity

Storage configurations for a data warehouse should be chosen based on the I/O bandwidth that they can provide, and not necessarily on their overall storage capacity. Buying storage based solely on capacity has the potential for making a mistake,

especially for systems less than 500GB is total size. The capacity of individual disk drives is growing faster than the I/O throughput rates provided by those disks, leading to a situation in which a small number of disks can store a large volume of data, but cannot provide the same I/O throughput as a larger number of small disks.

As an example, consider a 200GB data mart. Using 72GB drives, this data mart could be built with as few as six drives in a fully-mirrored environment. However, six drives might not provide enough I/O bandwidth to handle a medium number of concurrent users on a 4-CPU server. Thus, even though six drives provide sufficient storage, a larger number of drives may be required to provide acceptable performance for this system.

While it may not be practical to estimate the I/O bandwidth that is required by a data warehouse before a system is built, it is generally practical with the guidance of the hardware manufacturer to estimate how much I/O bandwidth a given server can potentially utilize, and ensure that the selected I/O configuration will be able to successfully feed the server. There are many variables in sizing the I/O systems, but one basic rule of thumb is that your data warehouse system should have multiple disks for each CPU (at least two disks for each CPU at a bare minimum) to achieve optimal performance.

Stripe Far and Wide

The guiding principle in configuring an I/O system for a data warehouse is to maximize I/O bandwidth by having multiple disks and channels access each database object. You can do this by striping the datafiles of the Oracle Database. A striped file is a file distributed across multiple disks. This striping can be managed by software (such as a logical volume manager), or within the storage hardware. The goal is to ensure that each tablespace is striped across a large number of disks (ideally, all of the disks) so that any database object can be accessed with the highest possible I/O bandwidth.

Use Redundancy

Because data warehouses are often the largest database systems in a company, they have the most disks and thus are also the most susceptible to the failure of a single disk. Therefore, disk redundancy is a requirement for data warehouses to protect against a hardware failure. Like disk-striping, redundancy can be achieved in many ways using software or hardware.

A key consideration is that occasionally a balance must be made between redundancy and performance. For example, a storage system in a RAID-5 configuration may be less expensive than a RAID-0+1 configuration, but it may not perform as well, either. Redundancy is necessary for any data warehouse, but the approach to redundancy may vary depending upon the performance and cost constraints of each data warehouse.

Test the I/O System Before Building the Database

The most important time to examine and tune the I/O system is before the database is even created. Once the database files are created, it is more difficult to reconfigure the files. Some logical volume managers may support dynamic reconfiguration of files, while other storage configurations may require that files be entirely rebuilt to reconfigure their I/O layout. In both cases, considerable system resources must be devoted to this reconfiguration.

When creating a data warehouse on a new system, the I/O bandwidth should be tested before creating all of the database datafiles to validate that the expected I/O levels are being achieved. On most operating systems, this can be done with simple scripts to measure the performance of reading and writing large test files.

Plan for Growth

A data warehouse designer should plan for future growth of a data warehouse. There are many approaches to handling the growth in a system, and the key consideration is to be able to grow the I/O system without compromising on the I/O bandwidth. You cannot, for example, add four disks to an existing system of 20 disks, and grow the database by adding a new tablespace striped across only the four new disks. A better solution would be to add new tablespaces striped across all 24 disks, and over time also convert the existing tablespaces striped across 20 disks to be striped across all 24 disks.

Storage Management

Two features to consider for managing disks are Oracle Managed Files and Automatic Storage Management. Without these features, a database administrator must manage the database files, which, in a data warehouse, can be hundreds or even thousands of files. Oracle Managed Files simplifies the administration of a database by providing functionality to automatically create and manage files, so the database administrator no longer needs to manage each database file. Automatic Storage Management provides additional functionality for managing not only files but also the disks. With Automatic Storage Management, the database administrator would administer a small number of disk groups. Automatic Storage Management handles the tasks of striping and providing disk redundancy, including rebalancing the database files when new disks are added to the system.

> **See Also:** *Oracle Database Storage Administrator's Guide* for more details

5

Partitioning in Data Warehouses

This chapter provides an introduction to the topic of partitioning in a data warehousing environment, and includes:

Overview of Partitioning in Data Warehouses

Data warehouses often contain very large tables and require techniques both for managing these large tables and for providing good query performance across them. An important tool for achieving this, as well as enhancing data access and improving overall application performance is partitioning.

Partitioning offers support for very large tables and indexes by letting you decompose them into smaller and more manageable pieces called partitions. This support is especially important for applications that access tables and indexes with millions of rows and many gigabytes of data. Partitioned tables and indexes facilitate administrative operations by enabling these operations to work on subsets of data. For example, you can add a new partition, organize an existing partition, or drop a partition with minimal to zero interruption to a read-only application.

Partitioning can help you tune SQL statements to avoid unnecessary index and table scans (using partition pruning). It also enables you to improve the performance of massive join operations when large amounts of data (for example, several million rows) are joined together by using partition-wise joins. Finally, partitioning data greatly improves manageability of very large databases and dramatically reduces the time required for administrative tasks such as backup and restore.

When adding or creating a partition, you have the option of deferring the segment creation until the data is first inserted, which is particularly valuable when installing applications that have a large footprint.

Granularity in a partitioning scheme can be easily changed by splitting or merging partitions. Thus, if a table's data is skewed to fill some partitions more than others, the ones that contain more data can be split to achieve a more even distribution. Partitioning also enables you to swap partitions with a table. By being able to easily add, remove, or swap a large amount of data quickly, swapping can be used to keep a large amount of data that is being loaded inaccessible until loading is completed, or can be used as a way to stage data between different phases of use. Some examples are current day's transactions or online archives.

A good starting point for considering partitioning strategies is to use the partitioning advice within the SQL Access Advisor, part of the Tuning Pack. The SQL Access Advisor offers both graphical and command-line interfaces.

A complementary approach that is commonly used with partitioning is parallel execution, which speeds up long-running queries, ETL, and some other operations. For data warehouses with very high loads of parallel statements, parallel statement queuing can be used to automatically manage the parallel statements.

See Also:

- Chapter 6, "Parallel Execution in Data Warehouses"
- *Oracle Database VLDB and Partitioning Guide* for a detailed examination of how and when to use partitioning as well as parallel execution and parallel statement queuing
- *Oracle Database 2 Day + Performance Tuning Guide* for details regarding the SQL Access Advisor

6

Parallel Execution in Data Warehouses

This chapter introduces the idea of parallel execution, which enables you to achieve good performance with data warehouses, and includes:

What is Parallel Execution?

Databases today, irrespective of whether they are data warehouses, operational data stores, or OLTP systems, contain a large amount of information. However, finding and presenting the right information in a timely fashion can be a challenge because of the vast quantity of data involved.

Parallel execution is the capability that addresses this challenge. Using parallel execution (also called parallelism), terabytes of data can be processed in minutes, not hours or days, simply by using multiple processes to accomplish a single task. This dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems (DSS) and data warehouses. You can also implement parallel execution on OLTP system for batch processing or schema maintenance operations such as index creation. Parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time. An example of this is when four processes combine to calculate the total sales for a year, each process handles one quarter of the year instead of a single processing handling all four quarters by itself. The improvement in performance can be quite significant.

Parallel execution improves processing for:

- Queries requiring large table scans, joins, or partitioned index scans
- Creations of large indexes
- Creation of large tables (including materialized views)
- Bulk inserts, updates, merges, and deletes

You can also use parallel execution to access object types within an Oracle database. For example, you can use parallel execution to access large objects (LOBs).

Large data warehouses should always use parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution.

Why Use Parallel Execution?

Imagine that your task is to count the number of cars in a street. There are two ways to do this. One, you can go through the street by yourself and count the number of cars or you can enlist a friend and then the two of you can start on opposite ends of the street, count cars until you meet each other and add the results of both counts to complete the task.

Assuming your friend counts equally fast as you do, you expect to complete the task of counting all cars in a street in roughly half the time compared to when you perform the job all by yourself. If this is the case, then your operations scales linearly. That is, twice the number of resources halves the total processing time.

A database is not very different from the counting cars example. If you allocate twice the number of resources and achieve a processing time that is half of what it was with the original amount of resources, then the operation scales linearly. Scaling linearly is the ultimate goal of parallel processing, both in counting cars as well as in delivering answers from a database query.

See Also:

- *Oracle Database Concepts* for a general introduction to parallelism concepts
- *Oracle Database VLDB and Partitioning Guide* for more information about using parallel execution

When to Implement Parallel Execution

Parallel execution benefits systems with all of the following characteristics:

- Symmetric multiprocessors (SMPs), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes, such as sorts, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution may reduce system performance on overutilized systems or systems with small I/O bandwidth.

The benefits of parallel execution can be seen in DSS and data warehousing environments. OLTP systems can also benefit from parallel execution during batch processing and during schema maintenance operations such as creation of indexes. The average simple DML or `SELECT` statements that characterize OLTP applications would not see any benefit from being executed in parallel.

When Not to Implement Parallel Execution

Parallel execution is not normally useful for:

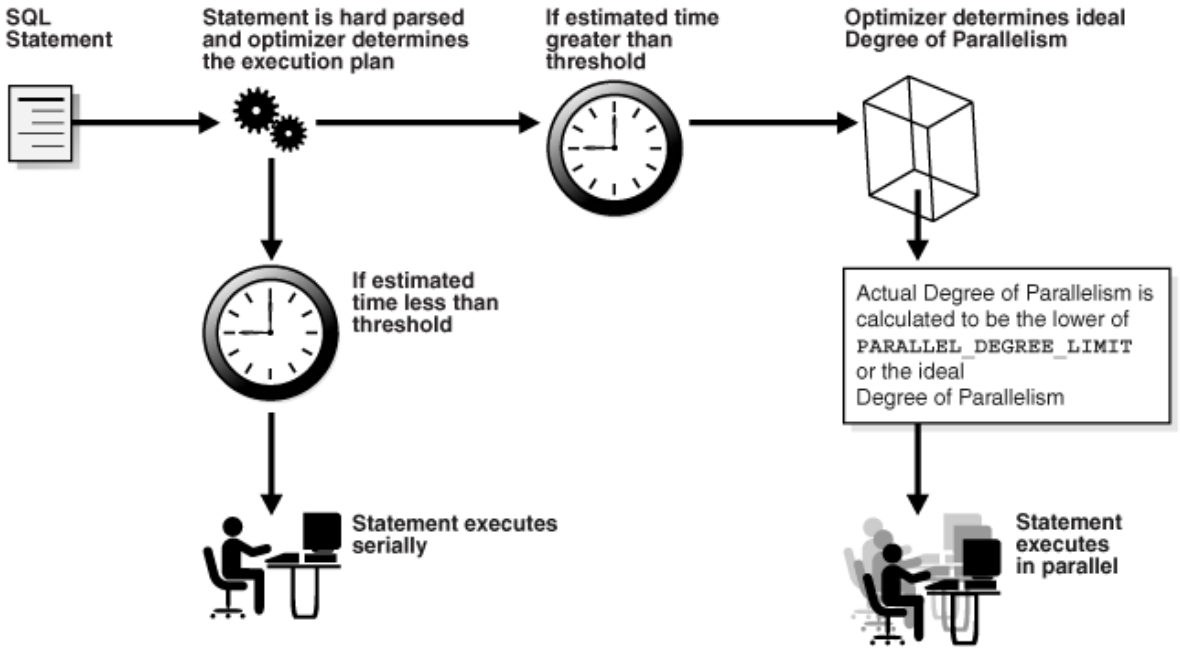
- Environments in which the typical query or transaction is very short (a few seconds or less). This includes most online transaction systems. Parallel execution is not useful in these environments because there is a cost associated with coordinating the parallel execution servers; for short transactions, the cost of this coordination may outweigh the benefits of parallelism.

- Environments in which the CPU, memory, or I/O resources are heavily utilized. Parallel execution is designed to exploit additional available hardware resources; if no such resources are available, then parallel execution does not yield any benefits and indeed may be detrimental to performance.

Automatic Degree of Parallelism and Statement Queuing

As the name implies, automatic degree of parallelism is where Oracle Database determines the degree of parallelism (DOP) with which to run a statement (DML, DDL, and queries) based on the fastest possible plan as determined by the optimizer. That means that the database parses a query, calculates the cost and then calculates a DOP to run with. The cheapest plan may be to run serially, which is also an option. Figure 6–1, "Optimizer Calculation: Serial or Parallel?" illustrates this decision making process.

Figure 6–1 Optimizer Calculation: Serial or Parallel?



Should you choose to use automatic DOP, you may see many more statements running in parallel, especially if the threshold is relatively low, where low is relative to the system and not an absolute quantifier.

Because of this expected behavior of more statements running in parallel with automatic DOP, it becomes more important to manage the utilization of the parallel processes available. That means that the system should be intelligent about when to run a statement and verify whether the requested numbers of parallel processes are available. The requested number of processes in this is the DOP for that statement.

The answer to this management question is parallel statement queuing with the Database Resource Manager. Parallel statement queuing runs a statement when its requested DOP is available. For example, when a statement requests a DOP of 64, it will not run if there are only 32 processes currently free to assist this customer, so the statement will be placed into a queue.

With Database Resource Manager, you can classify statements into workloads through consumer groups. Each consumer group can then be given the appropriate priority

and the appropriate levels of parallel processes. Each consumer group also has its own queue to queue parallel statements based on the system load.

See Also:

- *Oracle Database VLDB and Partitioning Guide* for more information about using automatic DOP with parallel execution
- *Oracle Database Administrator's Guide* for more information about using the Database Resource Manager

In-Memory Parallel Execution

Traditionally, parallel processing by-passed the database buffer cache for most operations, reading data directly from disk (through direct path I/O) into the parallel execution server's private working space. Only objects smaller than about 2% of `DB_CACHE_SIZE` would be cached in the database buffer cache of an instance, and most objects accessed in parallel are larger than this limit. This behavior meant that parallel processing rarely took advantage of the available memory other than for its private processing. However, over the last decade, hardware systems have evolved quite dramatically; the memory capacity on a typical database server is now in the double or triple digit gigabyte range. This, together with Oracle's compression technologies and the capability of Oracle Database 11*g* Release 2 to exploit the aggregated database buffer cache of an Oracle Real Application Clusters environment now enables caching of objects in the terabyte range.

In-Memory parallel execution takes advantage of this large aggregated database buffer cache. By having parallel execution servers access objects using the database buffer cache, they can scan data at least ten times faster than they can on disk.

With In-Memory parallel execution, when a SQL statement is issued in parallel, a check is conducted to determine if the objects accessed by the statement should be cached in the aggregated buffer cache of the system. In this context, an object can either be a table, index, or, in the case of partitioned objects, one or multiple partitions.

See Also:

- *Oracle Database VLDB and Partitioning Guide* for more information about using In-Memory parallel execution

7

Indexes

This chapter contains the following topics:

> **See Also:** *Oracle Database Concepts* for general information regarding indexing

Using Bitmap Indexes in Data Warehouses

Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries.
- Reduced storage requirements compared to other indexing techniques.
- Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory.
- Efficient maintenance during parallel DML and loads.

Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of disk space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of rowids for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key value replaces a list of rowids.

Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual rowid, so that the bitmap index provides the same functionality as a regular index. Bitmap indexes store the bitmaps in a compressed way. If the number of distinct key values is small, bitmap indexes compress better and the space saving benefit compared to a B-tree index becomes even better.

Bitmap indexes are most effective for queries that contain multiple conditions in the `WHERE` clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically. If you are

unsure of which indexes to create, the SQL Access Advisor can generate recommendations on what to create. As the bitmaps from bitmap indexes can be combined quickly, it is usually best to use single-column bitmap indexes.

When creating bitmap indexes, you should use `NOLOGGING` and `COMPUTE STATISTICS`. In addition, you should keep in mind that bitmap indexes are usually easier to destroy and re-create than to maintain.

Benefits for Data Warehousing Applications

Bitmap indexes are primarily intended for data warehousing applications where users query the data rather than update it. They are not suitable for OLTP applications with large numbers of concurrent transactions modifying the data.

Parallel query and parallel DML work with bitmap indexes. Bitmap indexing also supports parallel create indexes and concatenated indexes.

> **See Also:** Chapter 20, "Schema Modeling Techniques" for further information about using bitmap indexes in data warehousing environments

Cardinality

The advantages of using bitmap indexes are greatest for columns in which the ratio of the number of distinct values to the number of rows in the table is small. We refer to this ratio as the **degree of cardinality**. A gender column, which has only two distinct values (male and female), is optimal for a bitmap index. However, data warehouse administrators also build bitmap indexes on columns with higher cardinalities.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can outperform a B-tree index, particularly when this column is often queried in conjunction with other indexed columns. In fact, in a typical data warehouse environments, a bitmap index can be considered for any non-unique column.

B-tree indexes are most effective for high-cardinality data: that is, for data with many possible values, such as `customer_name` or `phone_number`. In a data warehouse, B-tree indexes should be used only for unique columns or other columns with very high cardinalities (that is, columns that are almost unique). The majority of indexes in a data warehouse should be bitmap indexes.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. `AND` and `OR` conditions in the `WHERE` clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to rowids. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

Example 7–1 Bitmap Index

The following shows a portion of a company's `customers` table.

```
SELECT cust_id, cust_gender, cust_marital_status, cust_income_level
FROM customers;

CUST_ID    C CUST_MARITAL_STATUS  CUST_INCOME_LEVEL
---------- - -------------------- ---------------------
...
        70 F                    D: 70,000 - 89,999
        80 F married            H: 150,000 - 169,999
        90 M single             H: 150,000 - 169,999
```

```
       100 F                      I: 170,000 - 189,999
       110 F married              C: 50,000 - 69,999
       120 M single               F: 110,000 - 129,999
       130 M                      J: 190,000 - 249,999
       140 M married              G: 130,000 - 149,999
...
```

Because cust_gender, cust_marital_status, and cust_income_level are all low-cardinality columns (there are only three possible values for marital status, two possible values for gender, and 12 for income level), bitmap indexes are ideal for these columns. Do not create a bitmap index on cust_id because this is a unique column. Instead, a unique B-tree index on this column provides the most efficient representation and retrieval.

Table 7–1 illustrates the bitmap index for the cust_gender column in this example. It consists of two separate bitmaps, one for gender.

Table 7–1 Sample Bitmap Index

	gender='M'	gender='F'
cust_id 70	0	1
cust_id 80	0	1
cust_id 90	1	0
cust_id 100	0	1
cust_id 110	0	1
cust_id 120	1	0
cust_id 130	1	0
cust_id 140	1	0

Each entry (or bit) in the bitmap corresponds to a single row of the customers table. The value of each bit depends upon the values of the corresponding row in the table. For example, the bitmap cust_gender='F' contains a one as its first bit because the gender is F in the first row of the customers table. The bitmap cust_gender='F' has a zero for its third bit because the gender of the third row is not F.

An analyst investigating demographic trends of the company's customers might ask, "How many of our married customers have an income level of G or H?" This corresponds to the following query:

```
SELECT COUNT(*) FROM customers
WHERE cust_marital_status = 'married'
AND cust_income_level IN ('H: 150,000 - 169,999', 'G: 130,000 - 149,999');
```

Bitmap indexes can efficiently process this query by merely counting the number of ones in the bitmap illustrated in Figure 7–1. The result set will be found by using bitmap OR merge operations without the necessity of a conversion to rowids. To identify additional specific customer attributes that satisfy the criteria, use the resulting bitmap to access the table after a bitmap to rowid conversion.

Figure 7–1 Executing a Query Using Bitmap Indexes

How to Determine Candidates for Using a Bitmap Index

Bitmap indexes should help when either the fact table is queried alone, and there are predicates on the indexed column, or when the fact table is joined with two or more dimension tables, and there are indexes on foreign key columns in the fact table, and predicates on dimension table columns.

A fact table column is a candidate for a bitmap index when the following conditions are met:

- There are 100 or more rows for each distinct value in the indexed column. When this limit is met, the bitmap index will be much smaller than a regular index, and you will be able to create the index much faster than a regular index. An example would be one million distinct values in a multi-billion row table.

And either of the following are true:

- The indexed column will be restricted in queries (referenced in the `WHERE` clause).

or

- The indexed column is a foreign key for a dimension table. In this case, such an index will make star transformation more likely.

Bitmap Indexes and Nulls

Unlike most other types of indexes, bitmap indexes include rows that have `NULL` values. Indexing of nulls can be useful for some types of SQL statements, such as queries with the aggregate function `COUNT`.

Example 7–2 Bitmap Index

```
SELECT COUNT(*) FROM customers WHERE cust_marital_status IS NULL;
```

This query uses a bitmap index on `cust_marital_status`. Note that this query would not be able to use a B-tree index, because B-tree indexes do not store the `NULL` values.

```
SELECT COUNT(*) FROM customers;
```

Any bitmap index can be used for this query because all table rows are indexed, including those that have `NULL` data. If nulls were not indexed, the optimizer would be able to use indexes only on columns with `NOT NULL` constraints.

Bitmap Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables but they must be local to the partitioned table—they cannot be global indexes. A partitioned table can only have global B-tree indexes, partitioned or nonpartitioned.

See Also:

- *Oracle Database VLDB and Partitioning Guide* for more information

Using Bitmap Join Indexes in Data Warehouses

In addition to a bitmap index on a single table, you can create a bitmap join index, which is a bitmap index for the join of two or more tables. In a bitmap join index, the bitmap for the table to be indexed is built for values coming from the joined tables. In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.

A bitmap join index can improve the performance by an order of magnitude. By storing the result of a join, the join can be avoided completely for SQL statements using a bitmap join index. Furthermore, since it is most likely to have a much smaller number of distinct values for a bitmap join index compared to a regular bitmap index on the join column, the bitmaps compress better, yielding to less space consumption than a regular bitmap index on the join column.

Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance. This is because the materialized join views do not compress the rowids of the fact tables.

B-tree and bitmap indexes have different maximum column limitations.

See Also:

- *Oracle Database SQL Language Reference* for details regarding these limitations

Four Join Models for Bitmap Join Indexes

The most common usage of a bitmap join index is in star model environments, where a large table is indexed on columns joined by one or several smaller tables. We will refer to the large table as the fact table and to the smaller tables as dimension tables. The following section describes the four different join models supported by bitmap join indexes. See Chapter 20, "Schema Modeling Techniques" for schema modeling techniques.

Example 7–3 Bitmap Join Index: One Dimension Table Columns Joins One Fact Table

Unlike the example in "Bitmap Index" on page 7-2, where a bitmap index on the `cust_gender` column on the `customers` table was built, we now create a bitmap join index on the fact table `sales` for the joined column `customers(cust_gender)`. Table `sales` stores `cust_id` values only:

```
SELECT time_id, cust_id, amount_sold FROM sales;

TIME_ID   CUST_ID    AMOUNT_SOLD
--------- ---------- -----------
01-JAN-98      29700        2291
01-JAN-98       3380         114
01-JAN-98      67830         553
01-JAN-98     179330           0
```

```
01-JAN-98     127520         195
01-JAN-98      33030         280
...
```

To create such a bitmap join index, column `customers(cust_gender)` has to be joined with table `sales`. The join condition is specified as part of the `CREATE` statement for the bitmap join index as follows:

```
CREATE BITMAP INDEX sales_cust_gender_bjix
ON sales(customers.cust_gender)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

The following query shows illustrates the join result that is used to create the bitmaps that are stored in the bitmap join index:

```
SELECT sales.time_id, customers.cust_gender, sales.amount_sold
FROM sales, customers
WHERE sales.cust_id = customers.cust_id;

TIME_ID   C AMOUNT_SOLD
--------- - -----------
01-JAN-98 M        2291
01-JAN-98 F         114
01-JAN-98 M         553
01-JAN-98 M           0
01-JAN-98 M         195
01-JAN-98 M         280
01-JAN-98 M          32
...
```

Table 7–2 illustrates the bitmap representation for the bitmap join index in this example.

Table 7–2 Sample Bitmap Join Index

	cust_gender='M'	cust_gender='F'
sales record 1	1	0
sales record 2	0	1
sales record 3	1	0
sales record 4	1	0
sales record 5	1	0
sales record 6	1	0
sales record 7	1	0

You can create other bitmap join indexes using more than one column or more than one table, as shown in these examples.

Example 7–4 Bitmap Join Index: Multiple Dimension Columns Join One Fact Table

You can create a bitmap join index on more than one column from a single dimension table, as in the following example, which uses `customers(cust_gender, cust_marital_status)` from the `sh` schema:

```
CREATE BITMAP INDEX sales_cust_gender_ms_bjix
ON sales(customers.cust_gender, customers.cust_marital_status)
```

```
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

Example 7–5 Bitmap Join Index: Multiple Dimension Tables Join One Fact Table

You can create a bitmap join index on multiple dimension tables, as in the following, which uses `customers(gender)` and `products(category)`:

```
CREATE BITMAP INDEX sales_c_gender_p_cat_bjix
ON sales(customers.cust_gender, products.prod_category)
FROM sales, customers, products
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id = products.prod_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

Example 7–6 Bitmap Join Index: Snowflake Schema

You can create a bitmap join index on more than one table, in which the indexed column is joined to the indexed table by using another table. For example, you can build an index on `countries.country_name`, even though the `countries` table is not joined directly to the `sales` table. Instead, the `countries` table is joined to the `customers` table, which is joined to the `sales` table. This type of schema is commonly called a **snowflake schema**.

```
CREATE BITMAP INDEX sales_co_country_name_bjix
ON sales(countries.country_name)
FROM sales, customers, countries
WHERE sales.cust_id = customers.cust_id
  AND customers.country_id = countries.country_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

Bitmap Join Index Restrictions and Requirements

Join results must be stored, therefore, bitmap join indexes have the following restrictions:

- Parallel DML is only supported on the fact table. Parallel DML on one of the participating dimension tables will mark the index as unusable.
- Only one table can be updated concurrently by different transactions when using the bitmap join index.
- No table can appear twice in the join.
- You cannot create a bitmap join index on a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension table join columns must be either primary key columns or have unique constraints.
- The dimension table column(s) participating the join with the fact table must be either the primary key column(s) or with the unique constraint.
- If a dimension table has composite primary key, each column in the primary key must be part of the join.
- The restrictions for creating a regular bitmap index also apply to a bitmap join index. For example, you cannot create a bitmap index with the `UNIQUE` attribute. See *Oracle Database SQL Language Reference* for other restrictions.

Using B-Tree Indexes in Data Warehouses

A B-tree index is organized like an upside-down tree. The bottom level of the index holds the actual data values and pointers to the corresponding rows, much as the index in a book has a page number associated with each index entry.

In general, use B-tree indexes when you know that your typical query refers to the indexed column and retrieves a few rows. In these queries, it is faster to find the rows by looking at the index. However, using the book index analogy, if you plan to look at every single topic in a book, you might not want to look in the index for the topic and then look up the page. It might be faster to read through every chapter in the book. Similarly, if you are retrieving most of the rows in a table, it might not make sense to look up the index to find the table rows. Instead, you might want to read or scan the table.

B-tree indexes are most commonly used in a data warehouse to enforce unique keys. In many cases, it may not even be necessary to index these columns in a data warehouse, because the uniqueness was enforced as part of the preceding ETL processing, and because typical data warehouse queries may not work better with such indexes. B-tree indexes are more common in environments using third normal form schemas. In general, bitmap indexes should be more common than B-tree indexes in most data warehouse environments.

B-tree and bitmap indexes have different maximum column limitations. See *Oracle Database SQL Language Reference* for these limitations.

Using Index Compression

Bitmap indexes are always stored in a patented, compressed manner without the need of any user intervention. B-tree indexes, however, can be stored specifically in a compressed manner to enable huge space savings, storing more keys in each index block, which also leads to less I/O and better performance.

Key compression lets you compress a B-tree index, which reduces the storage overhead of repeated values. In the case of a nonunique index, all index columns can be stored in a compressed format, whereas in the case of a unique index, at least one index column has to be stored uncompressed.

Generally, keys in an index have two pieces, a grouping piece and a unique piece. If the key is not defined to have a unique piece, Oracle provides one in the form of a rowid appended to the grouping piece. Key compression is a method of breaking off the grouping piece and storing it so it can be shared by multiple unique pieces. The cardinality of the chosen columns to be compressed determines the compression ratio that can be achieved. So, for example, if a unique index that consists of five columns provides the uniqueness mostly by the last two columns, it is most optimal to choose the three leading columns to be stored compressed. If you choose to compress four columns, the repetitiveness will be almost gone, and the compression ratio will be worse.

Although key compression reduces the storage requirements of an index, it can increase the CPU time required to reconstruct the key column values during an index scan. It also incurs some additional storage overhead, because every prefix entry has an overhead of four bytes associated with it.

Choosing Between Local Indexes and Global Indexes

B-tree indexes on partitioned tables can be global or local. With Oracle8*i* and earlier releases, Oracle recommended that global indexes not be used in data warehouse

environments because a partition DDL statement (for example, `ALTER TABLE ... DROP PARTITION`) would invalidate the entire index, and rebuilding the index is expensive. Since Oracle Database 10*g*, global indexes can be maintained without Oracle marking them as unusable after DDL. This enhancement makes global indexes more effective for data warehouse environments.

However, local indexes will be more common than global indexes. Global indexes should be used when there is a specific requirement which cannot be met by local indexes (for example, a unique index on a non-partitioning key, or a performance requirement).

Bitmap indexes on partitioned tables are always local.

8

Integrity Constraints

This chapter describes integrity constraints. It contains the following topics:

Why Integrity Constraints are Useful in a Data Warehouse

Integrity constraints provide a mechanism for ensuring that data conforms to guidelines specified by the database administrator. The most common types of constraints include:

- `UNIQUE` constraints

  To ensure that a given column is unique

- `NOT NULL` constraints

  To ensure that no null values are allowed

- `FOREIGN KEY` constraints

  To ensure that two keys share a primary key to foreign key relationship

Constraints can be used for these purposes in a data warehouse:

- Data cleanliness

  Constraints verify that the data in the data warehouse conforms to a basic level of data consistency and correctness, preventing the introduction of dirty data.

- Query optimization

  The Oracle Database utilizes constraints when optimizing SQL queries. Although constraints can be useful in many aspects of query optimization, constraints are particularly important for query rewrite of materialized views.

Unlike data in many relational database environments, data in a data warehouse is typically added or modified under controlled circumstances during the extraction, transformation, and loading (ETL) process. Multiple users normally do not update the data warehouse directly, as they do in an OLTP system.

> **See Also:** Chapter 12, "Overview of Extraction, Transformation, and Loading"

Overview of Constraint States

To understand how best to use constraints in a data warehouse, you should first understand the basic purposes of constraints. Some of these purposes are:

- Enforcement

  In order to use a constraint for enforcement, the constraint must be in the `ENABLE` state. An enabled constraint ensures that all data modifications upon a given table (or tables) satisfy the conditions of the constraints. Data modification operations which produce data that violates the constraint fail with a constraint violation error.

- Validation

  To use a constraint for validation, the constraint must be in the `VALIDATE` state. If the constraint is validated, then all data that currently resides in the table satisfies the constraint.

  Note that validation is independent of enforcement. Although the typical constraint in an operational system is both enabled and validated, any constraint could be validated but not enabled or vice versa (enabled but not validated). These latter two cases are useful for data warehouses.

- Belief

  In some cases, you will know that the conditions for a given constraint are true, so you do not need to validate or enforce the constraint. However, you may wish for the constraint to be present anyway to improve query optimization and performance. When you use a constraint in this way, it is called a belief or `RELY` constraint, and the constraint must be in the `RELY` state. The `RELY` state provides you with a mechanism for telling Oracle that a given constraint is believed to be true.

  Note that the `RELY` state only affects constraints that have not been validated.

Typical Data Warehouse Integrity Constraints

This section assumes that you are familiar with the typical use of constraints. That is, constraints that are both enabled and validated. For data warehousing, many users have discovered that such constraints may be prohibitively costly to build and maintain. The topics discussed are:

UNIQUE Constraints in a Data Warehouse

A `UNIQUE` constraint is typically enforced using a `UNIQUE` index. However, in a data warehouse whose tables can be extremely large, creating a unique index can be costly both in processing time and in disk space.

Suppose that a data warehouse contains a table sales, which includes a column sales_id. sales_id uniquely identifies a single sales transaction, and the data warehouse administrator must ensure that this column is unique within the data warehouse.

One way to create the constraint is as follows:

```
ALTER TABLE sales ADD CONSTRAINT sales_uk
UNIQUE (prod_id, cust_id, promo_id, channel_id, time_id);
```

By default, this constraint is both enabled and validated. Oracle implicitly creates a unique index on sales_id to support this constraint. However, this index can be problematic in a data warehouse for three reasons:

- The unique index can be very large, because the sales table can easily have millions or even billions of rows.
- The unique index is rarely used for query execution. Most data warehousing queries do not have predicates on unique keys, so creating this index will probably not improve performance.
- If sales is partitioned along a column other than sales_id, the unique index must be global. This can detrimentally affect all maintenance operations on the sales table.

A unique index is required for unique constraints to ensure that each individual row modified in the sales table satisfies the UNIQUE constraint.

For data warehousing tables, an alternative mechanism for unique constraints is illustrated in the following statement:

```
ALTER TABLE sales ADD CONSTRAINT sales_uk
UNIQUE (prod_id, cust_id, promo_id, channel_id, time_id) DISABLE VALIDATE;
```

This statement creates a unique constraint, but, because the constraint is disabled, a unique index is not required. This approach can be advantageous for many data warehousing environments because the constraint now ensures uniqueness without the cost of a unique index.

However, there are trade-offs for the data warehouse administrator to consider with DISABLE VALIDATE constraints. Because this constraint is disabled, no DML statements that modify the unique column are permitted against the sales table. You can use one of two strategies for modifying this table in the presence of a constraint:

- Use DDL to add data to this table (such as exchanging partitions). See the example in Chapter 16, "Maintaining the Data Warehouse".
- Before modifying this table, drop the constraint. Then, make all necessary data modifications. Finally, re-create the disabled constraint. Re-creating the constraint is more efficient than re-creating an enabled constraint. However, this approach does not guarantee that data added to the sales table while the constraint has been dropped is unique.

FOREIGN KEY Constraints in a Data Warehouse

In a star schema data warehouse, FOREIGN KEY constraints validate the relationship between the fact table and the dimension tables. A sample constraint might be:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
ENABLE VALIDATE;
```

However, in some situations, you may choose to use a different state for the FOREIGN KEY constraints, in particular, the ENABLE NOVALIDATE state. A data warehouse administrator might use an ENABLE NOVALIDATE constraint when either:

- The tables contain data that currently disobeys the constraint, but the data warehouse administrator wishes to create a constraint for future enforcement.
- An enforced constraint is required immediately.

Suppose that the data warehouse loaded new data into the fact tables every day, but refreshed the dimension tables only on the weekend. During the week, the dimension tables and fact tables may in fact disobey the FOREIGN KEY constraints. Nevertheless, the data warehouse administrator might wish to maintain the enforcement of this constraint to prevent any changes that might affect the FOREIGN KEY constraint outside of the ETL process. Thus, you can create the FOREIGN KEY constraints every night, after performing the ETL process, as shown in the following:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
ENABLE NOVALIDATE;
```

ENABLE NOVALIDATE can quickly create an enforced constraint, even when the constraint is believed to be true. Suppose that the ETL process verifies that a FOREIGN KEY constraint is true. Rather than have the database re-verify this FOREIGN KEY constraint, which would require time and database resources, the data warehouse administrator could instead create a FOREIGN KEY constraint using ENABLE NOVALIDATE.

RELY Constraints

The ETL process commonly verifies that certain constraints are true. For example, it can validate all of the foreign keys in the data coming into the fact table. This means that you can trust it to provide clean data, instead of implementing constraints in the data warehouse. You create a RELY constraint as follows:

```
ALTER TABLE sales ADD CONSTRAINT sales_time_fk
FOREIGN KEY (time_id) REFERENCES times (time_id)
RELY DISABLE NOVALIDATE;
```

This statement assumes that the primary key is in the RELY state. RELY constraints, even though they are not used for data validation, can:

- Enable more sophisticated query rewrites for materialized views. See Chapter 18, "Basic Query Rewrite" for further details.
- Enable other data warehousing tools to retrieve information regarding constraints directly from the Oracle data dictionary.

Creating a RELY constraint is inexpensive and does not impose any overhead during DML or load. Because the constraint is not being validated, no data processing is necessary to create it.

NOT NULL Constraints

When using query rewrite, you should consider whether NOT NULL constraints are required. The primary situation where you will need to use them is for join back query rewrite. See Chapter 19, "Advanced Query Rewrite" for further information regarding NOT NULL constraints when using query rewrite.

Integrity Constraints and Parallelism

All constraints can be validated in parallel. When validating constraints on very large tables, parallelism is often necessary to meet performance goals. The degree of parallelism for a given constraint operation is determined by the default degree of parallelism of the underlying table.

Integrity Constraints and Partitioning

You can create and maintain constraints before you partition the data. Later chapters discuss the significance of partitioning for data warehousing. Partitioning can improve constraint management just as it does to management of many other operations. For example, Chapter 16, "Maintaining the Data Warehouse" provides a scenario creating `UNIQUE` and `FOREIGN KEY` constraints on a separate staging table, and these constraints are maintained during the `EXCHANGE PARTITION` statement.

View Constraints

You can create constraints on views. The only type of constraint supported on a view is a `RELY` constraint.

This type of constraint is useful when queries typically access views instead of base tables, and the database administrator thus needs to define the data relationships between views rather than tables.

> **See Also:** Chapter 9, "Basic Materialized Views" and Chapter 18, "Basic Query Rewrite"

9

Basic Materialized Views

This chapter describes the use of materialized views. It contains the following topics:

Overview of Data Warehousing with Materialized Views

Typically, data flows from one or more online transaction processing (OLTP) database into a data warehouse on a monthly, weekly, or daily basis. The data is normally processed in a **staging file** before being added to the data warehouse. Data warehouses commonly range in size from tens of gigabytes to a few terabytes. Usually, the vast majority of the data is stored in a few very large fact tables.

One technique employed in data warehouses to improve performance is the creation of summaries. Summaries are special types of aggregate views that improve query execution times by precalculating expensive joins and aggregation operations prior to execution and storing the results in a table in the database. For example, you can create a summary table to contain the sums of sales by region and by product.

The summaries or aggregates that are referred to in this book and in literature on data warehousing are created in Oracle Database using a schema object called a **materialized view**. Materialized views can perform a number of roles, such as improving query performance or providing replicated data.

In the past, organizations using summaries spent a significant amount of time and effort creating summaries manually, identifying which summaries to create, indexing the summaries, updating them, and advising their users on which ones to use. The introduction of summary management eased the workload of the database administrator and meant the user no longer needed to be aware of the summaries that had been defined. The database administrator creates one or more materialized views, which are the equivalent of a summary. The end user queries the tables and views at the detail data level. The query rewrite mechanism in the Oracle server automatically rewrites the SQL query to use the summary tables. This mechanism reduces response time for returning results from the query. Materialized views within the data warehouse are transparent to the end user or to the database application.

Although materialized views are usually accessed through the query rewrite mechanism, an end user or database application can construct queries that directly access the materialized views. However, serious consideration should be given to whether users should be allowed to do this because any change to the materialized views affects the queries that reference them.

Materialized Views for Data Warehouses

In data warehouses, you can use materialized views to precompute and store aggregated data such as the sum of sales. Materialized views in these environments are often referred to as summaries, because they store summarized data. They can also be used to precompute joins with or without aggregations. A materialized view eliminates the overhead associated with expensive joins and aggregations for a large or important class of queries.

Materialized Views for Distributed Computing

In distributed environments, you can use materialized views to replicate data at distributed sites and to synchronize updates done at those sites with conflict resolution methods. These replica materialized views provide local access to data that otherwise would have to be accessed from remote sites. Materialized views are also useful in remote data marts. See *Oracle Database Advanced Replication* and *Oracle Database Heterogeneous Connectivity Administrator's Guide* for details on distributed and mobile computing.

Materialized Views for Mobile Computing

You can also use materialized views to download a subset of data from central servers to mobile clients, with periodic refreshes and updates between clients and the central servers.

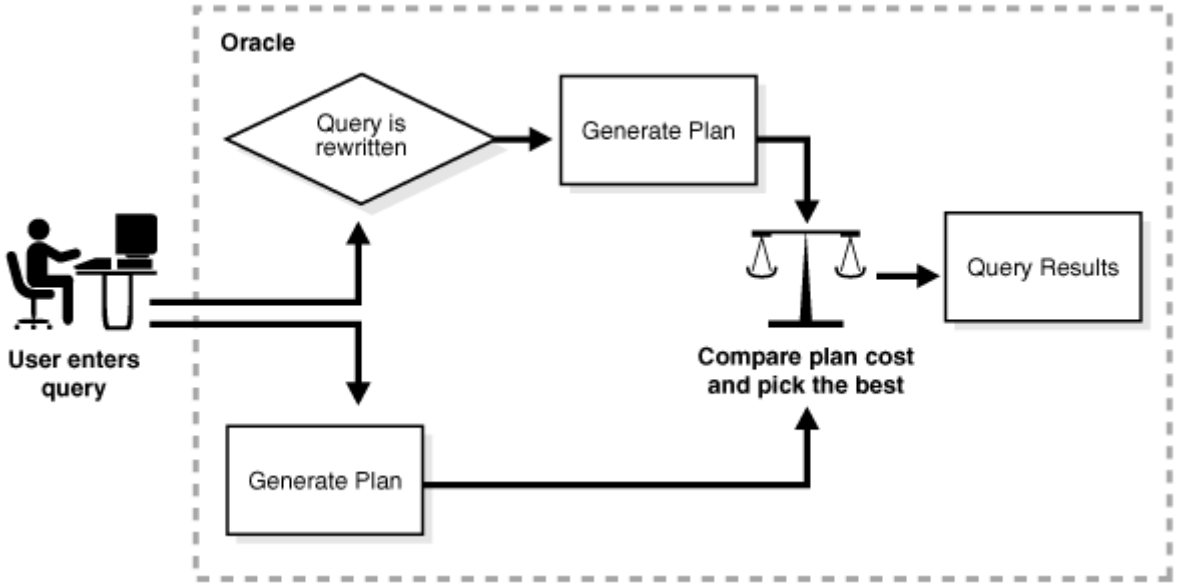
This chapter focuses on the use of materialized views in data warehouses. See *Oracle Database Advanced Replication* and *Oracle Database Heterogeneous Connectivity Administrator's Guide* for details on distributed and mobile computing.

The Need for Materialized Views

You can use materialized views to increase the speed of queries on very large databases. Queries to large databases often involve joins between tables, aggregations such as `SUM`, or both. These operations are expensive in terms of time and processing power. The type of materialized view you create determines how the materialized view is refreshed and used by query rewrite.

Materialized views improve query performance by precalculating expensive join and aggregation operations on the database prior to execution and storing the results in the database. The query optimizer automatically recognizes when an existing materialized view can and should be used to satisfy a request. It then transparently rewrites the request to use the materialized view. Queries go directly to the materialized view and not to the underlying detail tables. In general, rewriting queries to use materialized views rather than detail tables improves response time. Figure 9–1 illustrates how query rewrite works.

Figure 9–1 Transparent Query Rewrite



When using query rewrite, create materialized views that satisfy the largest number of queries. For example, if you identify 20 queries that are commonly applied to the detail or fact tables, then you might be able to satisfy them with five or six well-written materialized views. A materialized view definition can include any number of aggregations (`SUM`, `COUNT(x)`, `COUNT(*)`, `COUNT(DISTINCT x)`, `AVG`, `VARIANCE`, `STDDEV`, `MIN`, and `MAX`). It can also include any number of joins. If you are unsure of which materialized views to create, Oracle provides the SQL Access Advisor, which is a set of advisory procedures in the `DBMS_ADVISOR` package to help in designing and evaluating materialized views for query rewrite.

If a materialized view is to be used by query rewrite, it must be stored in the same database as the detail tables on which it relies. A materialized view can be partitioned, and you can define a materialized view on a partitioned table. You can also define one or more indexes on the materialized view.

Unlike indexes, materialized views can be accessed directly using a `SELECT` statement. However, it is recommended that you try to avoid writing SQL statements that directly reference the materialized view, because then it is difficult to change them without affecting the application. Instead, let query rewrite transparently rewrite your query to use the materialized view.

Note that the techniques shown in this chapter illustrate how to use materialized views in data warehouses. Materialized views can also be used by Oracle Replication. See *Oracle Database Advanced Replication* for further information.

Components of Summary Management

Summary management consists of:

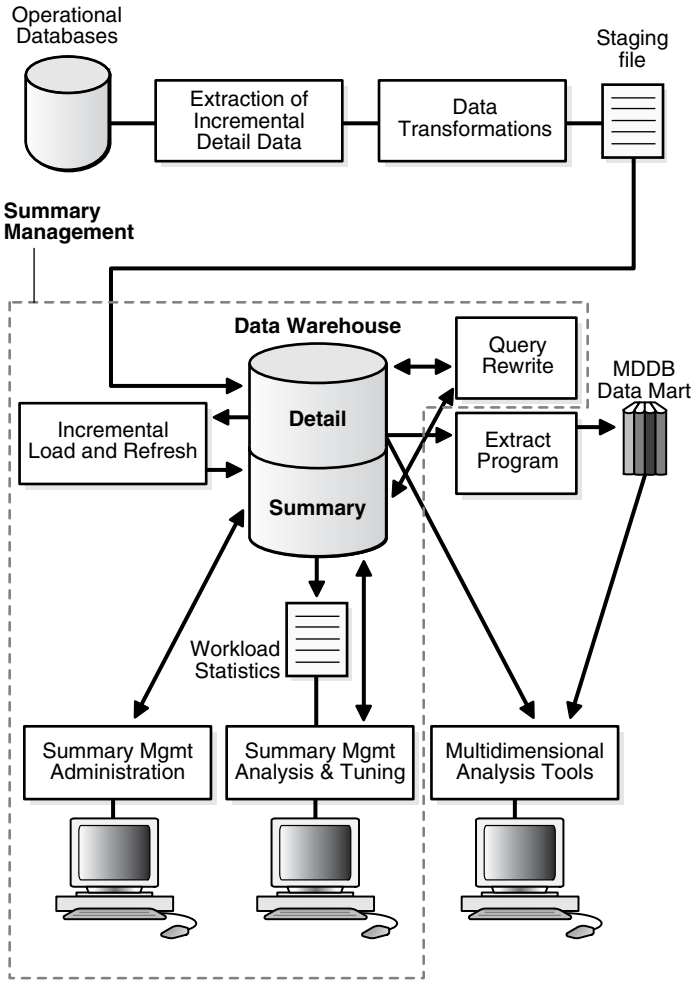
- Mechanisms to define materialized views and dimensions.
- A refresh mechanism to ensure that all materialized views contain the latest data.
- A query rewrite capability to transparently rewrite a query to use a materialized view.

- The SQL Access Advisor, which recommends materialized views, partitions, and indexes to create.
- `TUNE_MVIEW`, which shows you how to make your materialized view fast refreshable and use general query rewrite.

The use of summary management features imposes no schema restrictions, and can enable some existing DSS database applications to improve performance without the need to redesign the database or the application.

Figure 9–2 illustrates the use of summary management in the warehousing cycle. After the data has been transformed, staged, and loaded into the detail data in the warehouse, you can invoke the summary management process. First, use the SQL Access Advisor to plan how you will use materialized views. Then, create materialized views and design how queries will be rewritten. If you are having problems trying to get your materialized views to work then use `TUNE_MVIEW` to obtain an optimized materialized view.

Figure 9–2 Overview of Summary Management



Understanding the summary management process during the earliest stages of data warehouse design can yield large dividends later in the form of higher performance, lower summary administration costs, and reduced storage requirements.

Data Warehousing Terminology

Some basic data warehousing terms are defined as follows:

- **Dimension tables** describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

  Dimension tables usually change slowly over time and are not modified on a periodic schedule. They are used in long-running decision support queries to aggregate the data returned from the query into appropriate levels of the dimension hierarchy.

- **Hierarchies** describe the business relationships and common access patterns in the database. An analysis of the dimensions, combined with an understanding of the typical work load, can be used to create materialized views. See Chapter 11, "Dimensions" for more information.

- **Fact tables** describe the business transactions of an enterprise.

  The vast majority of data in a data warehouse is stored in a few very large fact tables that are updated periodically with data from one or more operational OLTP databases.

  Fact tables include facts (also called measures) such as sales, units, and inventory.

  - A simple measure is a numeric or character column of one table such as `fact.sales`.
  - A computed measure is an expression involving measures of one table, for example, `fact.revenues - fact.expenses`.
  - A multitable measure is a computed measure defined on multiple tables, for example, `fact_a.revenues - fact_b.expenses`.

  Fact tables also contain one or more foreign keys that organize the business transactions by the relevant business entities such as time, product, and market. In most cases, these foreign keys are non-null, form a unique compound key of the fact table, and each foreign key joins with exactly one row of a **dimension table**.

- A materialized view is a precomputed table comprising aggregated and joined data from fact and possibly from dimension tables. Among builders of data warehouses, a materialized view is also known as a **summary**.

Materialized View Schema Design

Summary management can perform many useful functions, including query rewrite and materialized view refresh, even if your data warehouse design does not follow these guidelines. However, you realize significantly greater query execution performance and materialized view refresh performance benefits and you require fewer materialized views if your schema design complies with these guidelines.

A materialized view definition includes any number of aggregates, as well as any number of joins. In several ways, a materialized view behaves like an index:

- The purpose of a materialized view is to increase query execution performance.
- The existence of a materialized view is transparent to SQL applications, so that a database administrator can create or drop materialized views at any time without affecting the validity of SQL applications.
- A materialized view consumes storage space.

- The contents of the materialized view must be updated when the underlying detail tables are modified.

Schemas and Dimension Tables

In the case of normalized or partially normalized dimension tables (a dimension that is stored in multiple tables), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. These relationships can be enabled with constraints, using the `NOVALIDATE` and `RELY` options if the relationships represented by the constraints are guaranteed by other means. Note that if the joins between fact and dimension tables do not support the parent-child relationship described previously, you still gain significant performance advantages from defining the dimension with the `CREATE DIMENSION` statement. Another alternative, subject to some restrictions, is to use outer joins in the materialized view definition (that is, in the `CREATE MATERIALIZED VIEW` statement).

You must not create dimensions in any schema that does not satisfy these relationships. Incorrect results can be returned from queries otherwise.

Materialized View Schema Design Guidelines

Before starting to define and use the various components of summary management, you should review your schema design to abide by the following guidelines wherever possible.

Guidelines 1 and 2 are more important than guideline 3. If your schema design does not follow guidelines 1 and 2, it does not then matter whether it follows guideline 3. Guidelines 1, 2, and 3 affect both query rewrite performance and materialized view refresh performance.

Table 9–1 Schema Design Guidelines

Schema Guideline	Description
Guideline 1 Dimensions	Dimensions should either be denormalized (each dimension contained in one table) or the joins between tables in a normalized or partially normalized dimension should guarantee that each child-side row joins with exactly one parent-side row. The benefits of maintaining this condition are described in "Creating Dimensions" on page 11-3. You can enforce this condition by adding `FOREIGN KEY` and `NOT NULL` constraints on the child-side join keys and `PRIMARY KEY` constraints on the parent-side join keys.
Guideline 2 Dimensions	If dimensions are denormalized or partially denormalized, hierarchical integrity must be maintained between the key columns of the dimension table. Each child key value must uniquely identify its parent key value, even if the dimension table is denormalized. Hierarchical integrity in a denormalized dimension can be verified by calling the `VALIDATE_DIMENSION` procedure of the `DBMS_DIMENSION` package.
Guideline 3 Dimensions	Fact and dimension tables should similarly guarantee that each fact table row joins with exactly one dimension table row. This condition must be declared, and optionally enforced, by adding `FOREIGN KEY` and `NOT NULL` constraints on the fact key column(s) and `PRIMARY KEY` constraints on the dimension key column(s), or by using outer joins. In a data warehouse, constraints are typically enabled with the `NOVALIDATE` and `RELY` clauses to avoid constraint enforcement performance overhead. See *Oracle Database SQL Language Reference* for further details.
Guideline 4 Incremental Loads	Incremental loads of your detail data should be done using the SQL*Loader direct-path option, or any bulk loader utility that uses Oracle's direct-path interface. This includes `INSERT ... AS SELECT` with the `APPEND` or `PARALLEL` hints, where the hints cause the direct loader log to be used during the insert. See *Oracle Database SQL Language Reference* and "Types of Materialized Views" on page 9-8 for more information.

Table 9–1 (Cont.) Schema Design Guidelines

Schema Guideline	Description
Guideline 5 Partitions	Range/composite partition your tables by a monotonically increasing time column if possible (preferably of type DATE).
Guideline 6 Dimensions	After each load and before refreshing your materialized view, use the VALIDATE_DIMENSION procedure of the DBMS_DIMENSION package to incrementally verify dimensional integrity.
Guideline 7 Time Dimensions	If a time dimension appears in the materialized view as a time column, partition and index the materialized view in the same manner as you have the fact tables.

If you are concerned with the time required to enable constraints and whether any constraints might be violated, then use the ENABLE NOVALIDATE with the RELY clause to turn on constraint checking without validating any of the existing constraints. The risk with this approach is that incorrect query results could occur if any constraints are broken. Therefore, as the designer, you must determine how clean the data is and whether the risk of incorrect results is too great.

Loading Data into Data Warehouses

A popular and efficient way to load data into a data warehouse or data mart is to use SQL*Loader with the DIRECT or PARALLEL option, Data Pump, or to use another loader tool that uses the Oracle direct-path API. See *Oracle Database Utilities* for the restrictions and considerations when using SQL*Loader with the DIRECT or PARALLEL keywords.

Loading strategies can be classified as one-phase or two-phase. In one-phase loading, data is loaded directly into the target table, quality assurance tests are performed, and errors are resolved by performing DML operations prior to refreshing materialized views. If a large number of deletions are possible, then storage utilization can be adversely affected, but temporary space requirements and load time are minimized.

In a two-phase loading process:

- Data is first loaded into a temporary table in the warehouse.
- Quality assurance procedures are applied to the data.
- Referential integrity constraints on the target table are disabled, and the local index in the target partition is marked unusable.
- The data is copied from the temporary area into the appropriate partition of the target table using INSERT AS SELECT with the PARALLEL or APPEND hint. The temporary table is then dropped. Alternatively, if the target table is partitioned, you can create a new (empty) partition in the target table and use ALTER TABLE ... EXCHANGE PARTITION to incorporate the temporary table into the target table. See *Oracle Database SQL Language Reference* for more information.
- The constraints are enabled, usually with the NOVALIDATE option.

Immediately after loading the detail data and updating the indexes on the detail data, the database can be opened for operation, if desired. You can disable query rewrite at the system level by issuing an ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE statement until all the materialized views are refreshed.

If QUERY_REWRITE_INTEGRITY is set to STALE_TOLERATED, access to the materialized view can be allowed at the session level to any users who do not require the materialized views to reflect the data from the latest load by issuing an ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE statement. This scenario does not apply when QUERY_REWRITE_INTEGRITY is either ENFORCED or TRUSTED because

the system ensures in these modes that only materialized views with updated data participate in a query rewrite.

Overview of Materialized View Management Tasks

The motivation for using materialized views is to improve performance, but the overhead associated with materialized view management can become a significant system management problem. When reviewing or evaluating some of the necessary materialized view management activities, consider some of the following:

- Identifying what materialized views to create initially.
- Indexing the materialized views.
- Ensuring that all materialized views and materialized view indexes are refreshed properly each time the database is updated.
- Checking which materialized views have been used.
- Determining how effective each materialized view has been on workload performance.
- Measuring the space being used by materialized views.
- Determining which new materialized views should be created.
- Determining which existing materialized views should be dropped.
- Archiving old detail and materialized view data that is no longer useful.

After the initial effort of creating and populating the data warehouse or data mart, the major administration overhead is the update process, which involves:

- Periodic extraction of incremental changes from the operational systems.
- Transforming the data.
- Verifying that the incremental changes are correct, consistent, and complete.
- Bulk-loading the data into the warehouse.
- Refreshing indexes and materialized views so that they are consistent with the detail data.

The update process must generally be performed within a limited period of time known as the **update window**. The update window depends on the **update frequency** (such as daily or weekly) and the nature of the business. For a daily update frequency, an update window of two to six hours might be typical.

You need to know your update window for the following activities:

- Loading the detail data
- Updating or rebuilding the indexes on the detail data
- Performing quality assurance tests on the data
- Refreshing the materialized views
- Updating the indexes on the materialized views

Types of Materialized Views

The `SELECT` clause in the materialized view creation statement defines the data that the materialized view is to contain. Only a few restrictions limit what can be specified. Any number of tables can be joined together. Besides tables, other elements such as

views, inline views (subqueries in the FROM clause of a SELECT statement), subqueries, and materialized views can all be joined or referenced in the SELECT clause. You cannot, however, define a materialized view with a subquery in the SELECT list of the defining query. You can, however, include subqueries elsewhere in the defining query, such as in the WHERE clause.

The types of materialized views are:

- Materialized Views with Aggregates
- Materialized Views Containing Only Joins
- Nested Materialized Views

Materialized Views with Aggregates

In data warehouses, materialized views normally contain aggregates as shown in Example 9–1. For fast refresh to be possible, the SELECT list must contain all of the GROUP BY columns (if present), and there must be a COUNT(*) and a COUNT(column) on any aggregated columns. Also, materialized view logs must be present on all tables referenced in the query that defines the materialized view. The valid aggregate functions are: SUM, COUNT(x), COUNT(*), AVG, VARIANCE, STDDEV, MIN, and MAX, and the expression to be aggregated can be any SQL value expression. See "Restrictions on Fast Refresh on Materialized Views with Aggregates" on page 9-21.

Fast refresh for a materialized view containing joins and aggregates is possible after any type of DML to the base tables (direct load or conventional INSERT, UPDATE, or DELETE). It can be defined to be refreshed ON COMMIT or ON DEMAND. A REFRESH ON COMMIT materialized view is refreshed automatically when a transaction that does DML to one of the materialized view's detail tables commits. The time taken to complete the commit may be slightly longer than usual when this method is chosen. This is because the refresh operation is performed as part of the commit process. Therefore, this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

Here are some examples of materialized views with aggregates. Note that materialized view logs are only created because this materialized view is fast refreshed.

Example 9–1 Example 1: Creating a Materialized View

```
CREATE MATERIALIZED VIEW LOG ON products WITH SEQUENCE, ROWID
(prod_id, prod_name, prod_desc, prod_subcategory, prod_subcategory_desc,
prod_category, prod_category_desc, prod_weight_class, prod_unit_of_measure,
 prod_pack_size, supplier_id, prod_status, prod_list_price, prod_min_price)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON sales
WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW product_sales_mv
PCTFREE 0  TABLESPACE demo
STORAGE (INITIAL 8M)
BUILD IMMEDIATE
REFRESH FAST
ENABLE QUERY REWRITE
AS SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales,
COUNT(*) AS cnt, COUNT(s.amount_sold) AS cnt_amt
```

```
FROM sales s, products p
WHERE s.prod_id = p.prod_id GROUP BY p.prod_name;
```

This example creates a materialized view product_sales_mv that computes total number and value of sales for a product. It is derived by joining the tables sales and products on the column prod_id. The materialized view is populated with data immediately because the build method is immediate and it is available for use by query rewrite. In this example, the default refresh method is FAST, which is allowed because the appropriate materialized view logs have been created on tables products and sales.

You can achieve better fast refresh performance for local materialized views if you use a materialized view log that contains a WITH COMMIT SCN clause. An example is the following:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID(prod_id, cust_id, time_id),
  COMMIT SCN INCLUDING NEW VALUES;
```

Example 9–2 Example 2: Creating a Materialized View

```
CREATE MATERIALIZED VIEW product_sales_mv
PCTFREE 0 TABLESPACE demo
STORAGE (INITIAL 8M)
BUILD DEFERRED
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_name;
```

This example creates a materialized view product_sales_mv that computes the sum of sales by prod_name. It is derived by joining the tables sales and products on the column prod_id. The materialized view does not initially contain any data, because the build method is DEFERRED. A complete refresh is required for the first refresh of a build deferred materialized view. When it is refreshed and once populated, this materialized view can be used by query rewrite.

Example 9–3 Example 3: Creating a Materialized View

```
CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sum_sales
PARALLEL
BUILD IMMEDIATE
REFRESH FAST ON COMMIT AS
SELECT s.prod_id, s.time_id, COUNT(*) AS count_grp,
   SUM(s.amount_sold) AS sum_dollar_sales,
   COUNT(s.amount_sold) AS count_dollar_sales,
   SUM(s.quantity_sold) AS sum_quantity_sales,
   COUNT(s.quantity_sold) AS count_quantity_sales
FROM sales s
GROUP BY s.prod_id, s.time_id;
```

This example creates a materialized view that contains aggregates on a single table. Because the materialized view log has been created with all referenced columns in the materialized view's defining query, the materialized view is fast refreshable. If DML is applied against the sales table, then the changes are reflected in the materialized view when the commit is issued.

Requirements for Using Materialized Views with Aggregates

Table 9–2 illustrates the aggregate requirements for materialized views. If aggregate X is present, aggregate Y is required and aggregate Z is optional.

Table 9–2 Requirements for Materialized Views with Aggregates

X	Y	Z
COUNT(expr)	-	-
MIN(expr)	-	-
MAX(expr)	-	-
SUM(expr)	COUNT(expr)	-
SUM(col), col has NOT NULL constraint	-	-
AVG(expr)	COUNT(expr)	SUM(expr)
STDDEV(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)
VARIANCE(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)

Note that COUNT(*) must always be present to guarantee all types of fast refresh. Otherwise, you may be limited to fast refresh after inserts only. Oracle recommends that you include the optional aggregates in column Z in the materialized view in order to obtain the most efficient and accurate fast refresh of the aggregates.

Materialized Views Containing Only Joins

Some materialized views contain only joins and no aggregates, such as in Example 9–4 on page 9-12, where a materialized view is created that joins the sales table to the times and customers tables. The advantage of creating this type of materialized view is that expensive joins are precalculated.

Fast refresh for a materialized view containing only joins is possible after any type of DML to the base tables (direct-path or conventional INSERT, UPDATE, or DELETE).

A materialized view containing only joins can be defined to be refreshed ON COMMIT or ON DEMAND. If it is ON COMMIT, the refresh is performed at commit time of the transaction that does DML on the materialized view's detail table.

If you specify REFRESH FAST, Oracle performs further verification of the query definition to ensure that fast refresh can be performed if any of the detail tables change. These additional checks are:

- A materialized view log must be present for each detail table unless the table supports PCT. Also, when a materialized view log is required, the ROWID column must be present in each materialized view log.
- The rowids of all the detail tables must appear in the SELECT list of the materialized view query definition.

If some of these restrictions are not met, you can create the materialized view as REFRESH FORCE to take advantage of fast refresh when it is possible. If one of the tables did not meet all of the criteria, but the other tables did, the materialized view would still be fast refreshable with respect to the other tables for which all the criteria are met.

To achieve an optimally efficient refresh, you should ensure that the defining query does not use an outer join that behaves like an inner join. If the defining query contains such a join, consider rewriting the defining query to contain an inner join. See "Restrictions on Fast Refresh on Materialized Views with Joins Only" on page 9-21 for more information regarding the conditions that cause refresh performance to degrade.

Materialized Join Views FROM Clause Considerations

If the materialized view contains only joins, the ROWID columns for each table (and each instance of a table that occurs multiple times in the FROM list) must be present in the SELECT list of the materialized view.

If the materialized view has remote tables in the FROM clause, all tables in the FROM clause must be located on that same site. Further, ON COMMIT refresh is not supported for materialized view with remote tables. Except for SCN-based materialized view logs, materialized view logs must be present on the remote site for each detail table of the materialized view and ROWID columns must be present in the SELECT list of the materialized view, as shown in the following example.

Example 9–4 Materialized View Containing Only Joins

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;
CREATE MATERIALIZED VIEW detail_sales_mv
PARALLEL BUILD IMMEDIATE
REFRESH FAST AS
SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "customers_rid",
       c.cust_id, c.cust_last_name, s.amount_sold, s.quantity_sold, s.time_id
FROM sales s, times t, customers c
WHERE  s.cust_id = c.cust_id(+) AND s.time_id = t.time_id(+);
```

Alternatively, if the previous example did not include the columns times_rid and customers_rid, and if the refresh method was REFRESH FORCE, then this materialized view would be fast refreshable only if the sales table was updated but not if the tables times or customers were updated.

```
CREATE MATERIALIZED VIEW detail_sales_mv
PARALLEL
BUILD IMMEDIATE
REFRESH FORCE AS
SELECT s.rowid "sales_rid", c.cust_id, c.cust_last_name, s.amount_sold,
   s.quantity_sold, s.time_id
FROM sales s, times t, customers c
WHERE s.cust_id = c.cust_id(+) AND s.time_id = t.time_id(+);
```

Nested Materialized Views

A nested materialized view is a materialized view whose definition is based on another materialized view. A nested materialized view can reference other relations in the database in addition to referencing materialized views.

Why Use Nested Materialized Views?

In a data warehouse, you typically create many aggregate views on a single join (for example, rollups along different dimensions). Incrementally maintaining these distinct materialized aggregate views can take a long time, because the underlying join has to be performed many times.

Using nested materialized views, you can create multiple single-table materialized views based on a joins-only materialized view and the join is performed just once. In addition, optimizations can be performed for this class of single-table aggregate materialized view and thus refresh is very efficient.

Example 9–5 Nested Materialized View

You can create a nested materialized view on materialized views, but all parent and base materialized views must contain joins or aggregates. If the defining queries for a materialized view do not contain joins or aggregates, it cannot be nested. All the underlying objects (materialized views or tables) on which the materialized view is defined must have a materialized view log. All the underlying objects are treated as if they were tables. In addition, you can use all the existing options for materialized views.

Using the tables and their columns from the `sh` sample schema, the following materialized views illustrate how nested materialized views can be created.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON times WITH ROWID;

/*create materialized view join_sales_cust_time as fast refreshable at
   COMMIT time */
CREATE MATERIALIZED VIEW join_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT c.cust_id, c.cust_last_name, s.amount_sold, t.time_id,
       t.day_number_in_week, s.rowid srid, t.rowid trid, c.rowid crid
FROM sales s, customers c, times t
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id;
```

To create a nested materialized view on the table `join_sales_cust_time`, you would have to create a materialized view log on the table. Because this will be a single-table aggregate materialized view on `join_sales_cust_time`, you must log all the necessary columns and use the `INCLUDING NEW VALUES` clause.

```
/* create materialized view log on join_sales_cust_time */
CREATE MATERIALIZED VIEW LOG ON join_sales_cust_time
WITH ROWID (cust_last_name, day_number_in_week, amount_sold),
INCLUDING NEW VALUES;

/* create the single-table aggregate materialized view sum_sales_cust_time
on join_sales_cust_time as fast refreshable at COMMIT time */
CREATE MATERIALIZED VIEW sum_sales_cust_time
REFRESH FAST ON COMMIT AS
SELECT COUNT(*) cnt_all, SUM(amount_sold) sum_sales, COUNT(amount_sold)
       cnt_sales, cust_last_name, day_number_in_week
FROM join_sales_cust_time
GROUP BY cust_last_name, day_number_in_week;
```

Nesting Materialized Views with Joins and Aggregates

Some types of nested materialized views cannot be fast refreshed. Use `EXPLAIN_MVIEW` to identify those types of materialized views. You can refresh a tree of nested materialized views in the appropriate dependency order by specifying the `nested = TRUE` parameter with the `DBMS_MVIEW.REFRESH` parameter. For example, if you call `DBMS_MVIEW.REFRESH ('SUM_SALES_CUST_TIME', nested => TRUE)`, the `REFRESH` procedure will first refresh the `join_sales_cust_time` materialized view, and then refresh the `sum_sales_cust_time` materialized view.

Nested Materialized View Usage Guidelines

You should keep the following in mind when deciding whether to use nested materialized views:

- If you want to use fast refresh, you should fast refresh all the materialized views along any chain.
- If you want the highest level materialized view to be fresh with respect to the detail tables, you must ensure that all materialized views in a tree are refreshed in the correct dependency order before refreshing the highest-level. You can automatically refresh intermediate materialized views in a nested hierarchy using the `nested = TRUE` parameter, as described in "Nesting Materialized Views with Joins and Aggregates" on page 9-13. If you do not specify `nested = TRUE` and the materialized views under the highest-level materialized view are stale, refreshing only the highest-level will succeed, but makes it fresh only with respect to its underlying materialized view, not the detail tables at the base of the tree.
- When refreshing materialized views, you must ensure that all materialized views in a tree are refreshed. If you only refresh the highest-level materialized view, the materialized views under it will be stale and you must explicitly refresh them. If you use the `REFRESH` procedure with the `nested` parameter value set to `TRUE`, only specified materialized views and their child materialized views in the tree are refreshed, and not their top-level materialized views. Use the `REFRESH_DEPENDENT` procedure with the nested parameter value set to `TRUE` if you want to ensure that all materialized views in a tree are refreshed.
- Freshness of a materialized view is calculated relative to the objects directly referenced by the materialized view. When a materialized view references another materialized view, the freshness of the topmost materialized view is calculated relative to changes in the materialized view it directly references, not relative to changes in the tables referenced by the materialized view it references.

Restrictions When Using Nested Materialized Views

You cannot create both a materialized view and a prebuilt materialized view on the same table. For example, If you have a table `costs` with a materialized view `cost_mv` based on it, you cannot then create a prebuilt materialized view on table `costs`. The result would make `cost_mv` a nested materialized view and this method of conversion is not supported.

Creating Materialized Views

A materialized view can be created with the `CREATE MATERIALIZED VIEW` statement or using Enterprise Manager. Example 9–6 illustrates creating an materialized view called `cust_sales_mv`.

Example 9–6 Creating a Materialized View

```
CREATE MATERIALIZED VIEW cust_sales_mv
PCTFREE 0 TABLESPACE demo
STORAGE (INITIAL 8M)
PARALLEL
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE AS
SELECT  c.cust_last_name, SUM(amount_sold) AS sum_amount_sold
FROM customers c, sales s WHERE s.cust_id = c.cust_id
GROUP BY c.cust_last_name;
```

It is not uncommon in a data warehouse to have already created summary or aggregation tables, and you might not wish to repeat this work by building a new materialized view. In this case, the table that already exists in the database can be registered as a prebuilt materialized view. This technique is described in "Registering Existing Materialized Views" on page 9-27.

Once you have selected the materialized views you want to create, follow these steps for each materialized view.

1. Design the materialized view. Existing user-defined materialized views do not require this step. If the materialized view contains many rows, then, if appropriate, the materialized view should be partitioned (if possible) and should match the partitioning of the largest or most frequently updated detail or fact table (if possible). Refresh performance benefits from partitioning, because it can take advantage of parallel DML capabilities and possible PCT-based refresh.
2. Use the `CREATE MATERIALIZED VIEW` statement to create and, optionally, populate the materialized view. If a user-defined materialized view already exists, then use the `ON PREBUILT TABLE` clause in the `CREATE MATERIALIZED VIEW` statement. Otherwise, use the `BUILD IMMEDIATE` clause to populate the materialized view immediately, or the `BUILD DEFERRED` clause to populate the materialized view later. A `BUILD DEFERRED` materialized view is disabled for use by query rewrite until the first `COMPLETE REFRESH`, after which it is automatically enabled, provided the `ENABLE QUERY REWRITE` clause has been specified.

> **See Also:** *Oracle Database SQL Language Reference* for descriptions of the SQL statements `CREATE MATERIALIZED VIEW`, `ALTER MATERIALIZED VIEW`, and `DROP MATERIALIZED VIEW`

Creating Materialized Views with Column Alias Lists

Currently, when a materialized view is created, if its defining query contains same-name columns in the `SELECT` list, the name conflicts need to be resolved by specifying unique aliases for those columns. Otherwise, the `CREATE MATERIALIZED VIEW` statement fails with the error messages of columns ambiguously defined. However, the standard method of attaching aliases in the `SELECT` clause for name resolution restricts the use of the full text match query rewrite and it will occur only when the text of the materialized view's defining query and the text of user input query are identical. Thus, if the user specifies select aliases in the materialized view's defining query while there is no alias in the query, the full text match comparison fails. This is particularly a problem for queries from Discoverer, which makes extensive use of column aliases.

The following is an example of the problem. `sales_mv` is created with column aliases in the `SELECT` clause but the input query `Q1` does not have the aliases. The full text match rewrite fails. The materialized view is as follows:

```
CREATE MATERIALIZED VIEW sales_mv
ENABLE QUERY REWRITE AS
SELECT s.time_id sales_tid, c.time_id costs_tid
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

Input query statement `Q1` is as follows:

```
SELECT s.time_id, c1.time_id
FROM sales s, products p, costs c1
```

```
WHERE s.prod_id = p.prod_id AND c1.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

Even though the materialized view's defining query is almost identical and logically equivalent to the user's input query, query rewrite does not happen because of the failure of full text match that is the only rewrite possibility for some queries (for example, a subquery in the `WHERE` clause).

You can add a column alias list to a `CREATE MATERIALIZED VIEW` statement. The column alias list explicitly resolves any column name conflict without attaching aliases in the `SELECT` clause of the materialized view. The syntax of the materialized view column alias list is illustrated in the following example:

```
CREATE MATERIALIZED VIEW sales_mv (sales_tid, costs_tid)
ENABLE QUERY REWRITE AS
SELECT s.time_id, c.time_id
FROM sales s, products p, costs c
WHERE s.prod_id = p.prod_id AND c.prod_id = p.prod_id AND
      p.prod_name IN (SELECT prod_name FROM products);
```

In this example, the defining query of `sales_mv` now matches exactly with the user query `Q1`, so full text match rewrite takes place.

Note that when aliases are specified in both the `SELECT` clause and the new alias list clause, the alias list clause supersedes the ones in the `SELECT` clause.

Naming Materialized Views

The name of a materialized view must conform to standard Oracle naming conventions. However, if the materialized view is based on a user-defined prebuilt table, then the name of the materialized view must exactly match that table name.

If you already have a naming convention for tables and indexes, you might consider extending this naming scheme to the materialized views so that they are easily identifiable. For example, instead of naming the materialized view `sum_of_sales`, it could be called `sum_of_sales_mv` to denote that this is a materialized view and not a table or view.

Storage And Table Compression

Unless the materialized view is based on a user-defined prebuilt table, it requires and occupies storage space inside the database. Therefore, the storage needs for the materialized view should be specified in terms of the tablespace where it is to reside and the size of the extents.

If you do not know how much space the materialized view requires, then the `DBMS_MVIEW.ESTIMATE_MVIEW_SIZE` package can estimate the number of bytes required to store this uncompressed materialized view. This information can then assist the design team in determining the tablespace in which the materialized view should reside.

You should use table compression with highly redundant data, such as tables with many foreign keys. This is particularly useful for materialized views created with the `ROLLUP` clause. Table compression reduces disk use and memory use (specifically, the buffer cache), often leading to a better scaleup for read-only operations. Table compression can also speed up query execution at the expense of update cost.

Table compression has been extended in this release with Hybrid Columnar Compression. Hybrid Columnar Compression, a feature of certain Oracle storage systems, utilizes a combination of both row and columnar methods for storing data.

When data is loaded, groups of rows are stored in columnar format, with the values for a given column stored and compressed together. Storing column data together, with the same data type and similar characteristics, drastically increases the storage savings achieved from compression. Hybrid Columnar Compression provides multiple levels of compression and is best suited for tables or partitions with minimal update activity.

See Also:

- *Oracle Database VLDB and Partitioning Guide* for more information about table compression
- *Oracle Database Administrator's Guide* for more information about table compression
- *Oracle Database SQL Language Reference* for a complete description of `STORAGE` semantics
- *Oracle Database Concepts* for more information about Hybrid Columnar Compression

Build Methods

Two build methods are available for creating the materialized view, as shown in Table 9–3. If you select `BUILD IMMEDIATE`, the materialized view definition is added to the schema objects in the data dictionary, and then the fact or detail tables are scanned according to the `SELECT` expression and the results are stored in the materialized view. Depending on the size of the tables to be scanned, this build process can take a considerable amount of time.

An alternative approach is to use the `BUILD DEFERRED` clause, which creates the materialized view without data, thereby enabling it to be populated at a later date using the `DBMS_MVIEW.REFRESH` package described in Chapter 16, "Maintaining the Data Warehouse".

Table 9–3 Build Methods

Build Method	Description
`BUILD IMMEDIATE`	Create the materialized view and then populate it with data.
`BUILD DEFERRED`	Create the materialized view definition but do not populate it with data.

Enabling Query Rewrite

Before creating a materialized view, you can verify what types of query rewrite are possible by calling the procedure `DBMS_MVIEW.EXPLAIN_MVIEW`, or use `DBMS_ADVISOR.TUNE_MVIEW` to optimize the materialized view so that many types of query rewrite are possible. Once the materialized view has been created, you can use `DBMS_MVIEW.EXPLAIN_REWRITE` to find out if (or why not) it will rewrite a specific query.

Even though a materialized view is defined, it will not automatically be used by the query rewrite facility. Even though query rewrite is enabled by default, you also must specify the `ENABLE QUERY REWRITE` clause if the materialized view is to be considered available for rewriting queries.

If this clause is omitted or specified as `DISABLE QUERY REWRITE` when the materialized view is created, the materialized view can subsequently be enabled for query rewrite with the `ALTER MATERIALIZED VIEW` statement.

If you define a materialized view as BUILD DEFERRED, it is not eligible for query rewrite until it is populated with data through a complete refresh.

Query Rewrite Restrictions

Query rewrite is not possible with all materialized views. If query rewrite is not occurring when expected, DBMS_MVIEW.EXPLAIN_REWRITE can help provide reasons why a specific query is not eligible for rewrite. If this shows that not all types of query rewrite are possible, use the procedure DBMS_ADVISOR.TUNE_MVIEW to see if the materialized view can be defined differently so that query rewrite is possible. Also, check to see if your materialized view satisfies all of the following conditions.

Materialized View Restrictions

You should keep in mind the following restrictions:

- The defining query of the materialized view cannot contain any non-repeatable expressions (ROWNUM, SYSDATE, non-repeatable PL/SQL functions, and so on).
- The query cannot contain any references to LONG or LONG RAW datatypes or object REFs.
- If the materialized view was registered as PREBUILT, the precision of the columns must agree with the precision of the corresponding SELECT expressions unless overridden by the WITH REDUCED PRECISION clause.

See Also:

- Chapter 19, "Advanced Query Rewrite"
- *Oracle Database SQL Language Reference*

General Query Rewrite Restrictions

You should keep in mind the following restrictions:

- A query can reference both local and remote tables. Such a query can be rewritten as long as an eligible materialized view referencing the same tables is available locally.
- Neither the detail tables nor the materialized view can be owned by SYS.
- If a column or expression is present in the GROUP BY clause of the materialized view, it must also be present in the SELECT list.
- Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as AVG(AVG(x)) or AVG(x)+AVG(x) are not allowed.
- CONNECT BY clauses are not allowed.

See Also:

- Chapter 19, "Advanced Query Rewrite"
- *Oracle Database SQL Language Reference*

Refresh Options

When you define a materialized view, you can specify three refresh options: how to refresh, what type of refresh, and can trusted constraints be used. If unspecified, the defaults are assumed as ON DEMAND, FORCE, and ENFORCED constraints respectively.

The two refresh execution modes are ON COMMIT and ON DEMAND. Depending on the materialized view you create, some options may not be available. Table 9–4 describes the refresh modes.

Table 9–4 Refresh Modes

Refresh Mode	Description
ON COMMIT	Refresh occurs automatically when a transaction that modified one of the materialized view's detail tables commits. This can be specified as long as the materialized view is fast refreshable (in other words, not complex). The ON COMMIT privilege is necessary to use this mode.
ON DEMAND	Refresh occurs when a user manually executes one of the available refresh procedures contained in the DBMS_MVIEW package (REFRESH, REFRESH_ALL_MVIEWS, REFRESH_DEPENDENT).

When a materialized view is maintained using the ON COMMIT method, the time required to complete the commit may be slightly longer than usual. This is because the refresh operation is performed as part of the commit process. Therefore this method may not be suitable if many users are concurrently changing the tables upon which the materialized view is based.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and that materialized view includes joins and aggregation, Oracle recommends you use ON COMMIT fast refresh rather than ON DEMAND fast refresh.

If you think the materialized view did not refresh, check the alert log or trace file.

If a materialized view fails during refresh at COMMIT time, you must explicitly invoke the refresh procedure using the DBMS_MVIEW package after addressing the errors specified in the trace files. Until this is done, the materialized view will no longer be refreshed automatically at commit time.

You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options: COMPLETE, FAST, FORCE, and NEVER. Table 9–5 describes the refresh options.

Table 9–5 Refresh Options

Refresh Option	Description
COMPLETE	Refreshes by recalculating the materialized view's defining query.
FAST	Applies incremental changes to refresh the materialized view using the information logged in the materialized view logs, or from a SQL*Loader direct-path or a partition maintenance operation.
FORCE	Applies FAST refresh if possible; otherwise, it applies COMPLETE refresh.
NEVER	Indicates that the materialized view will not be refreshed with refresh mechanisms.

Whether the fast refresh option is available depends upon the type of materialized view. You can call the procedure DBMS_MVIEW.EXPLAIN_MVIEW to determine whether fast refresh is possible.

You can also specify if it is acceptable to use trusted constraints and QUERY_REWRITE_INTEGRITY = TRUSTED during refresh. Any nonvalidated RELY constraint is a trusted constraint. For example, nonvalidated foreign key/primary key relationships, functional dependencies defined in dimensions or a materialized view in the UNKNOWN state. If query rewrite is enabled during refresh, these can improve the performance of refresh by enabling more performant query rewrites. Any materialized view that can use TRUSTED constraints for refresh is left in a state of trusted freshness (the UNKNOWN state) after refresh.

This is reflected in the column STALENESS in the view USER_MVIEWS. The column UNKNOWN_TRUSTED_FD in the same view is also set to Y, which means yes.

You can define this property of the materialized view either during create time by specifying REFRESH USING TRUSTED [ENFORCED] CONSTRAINTS or by using ALTER MATERIALIZED VIEW DDL.

Table 9–6 Constraints

Constraints to Use	Description
TRUSTED CONSTRAINTS	Refresh can use trusted constraints and QUERY_REWRITE_INTEGRITY = TRUSTED during refresh. This allows use of non-validated RELY constraints and rewrite against materialized views in UNKNOWN or FRESH state during refresh. The USING TRUSTED CONSTRAINTS clause enables you to create a materialized view on top of a table that has a non-NULL Virtual Private Database (VPD) policy on it. In this case, ensure that the materialized view behaves correctly. Materialized view results are computed based on the rows and columns filtered by VPD policy. Therefore, you must coordinate the materialized view definition with the VPD policy to ensure the correct results. Without the USING TRUSTED CONSTRAINTS clause, any VPD policy on a base table will prevent a materialized view from being created.
ENFORCED CONSTRAINTS	Refresh can use validated constraints and QUERY_REWRITE_INTEGRITY = ENFORCED during refresh. This allows use of only validated, enforced constraints and rewrite against materialized views in FRESH state during refresh.

The fast refresh of a materialized view is optimized using the available primary and foreign key constraints on the join columns. This foreign key/primary key optimization can significantly improve refresh performance. For example, for a materialized view that contains a join between a fact table and a dimension table, if only new rows were inserted into the dimension table with no change to the fact table since the last refresh, then there will be nothing to refresh for this materialized view. The reason is that, because of the primary key constraint on the join column(s) of the dimension table and foreign key constraint on the join column(s) of the fact table, the new rows inserted into the dimension table will not join with any fact table rows, thus there is nothing to refresh. Another example of this refresh optimization is when both the fact and dimension tables have inserts since the last refresh. In this case, Oracle Database will only perform a join of delta fact table with the dimension table. Without the foreign key/primary key optimization, two joins during the refresh would be required, a join of delta fact with the dimension table, plus a join of delta dimension with an image of the fact table from before the inserts.

Note that this optimized fast refresh using primary and foreign key constraints on the join columns is available with and without constraint enforcement. In the first case, primary and foreign key constraints are enforced by the Oracle Database. This, however, incurs the cost of constraint maintenance. In the second case, the application guarantees primary and foreign key relationships so the constraints are declared RELY NOVALIDATE and the materialized view is defined with the REFRESH FAST USING TRUSTED CONSTRAINTS option.

> **See Also:** *Oracle Database SQL Language Reference* for more detailed information regarding refresh restrictions

General Restrictions on Fast Refresh

The defining query of the materialized view is restricted as follows:

- The materialized view must not contain references to non-repeating expressions like SYSDATE and ROWNUM.
- The materialized view must not contain references to RAW or LONG RAW data types.
- It cannot contain a SELECT list subquery.

- It cannot contain analytic functions (for example, RANK) in the SELECT clause.
- It cannot contain a MODEL clause.
- It cannot contain a HAVING clause with a subquery.
- It cannot contain nested queries that have ANY, ALL, or NOT EXISTS.
- It cannot contain a [START WITH …] CONNECT BY clause.
- It cannot contain multiple detail tables at different sites.
- ON COMMIT materialized views cannot have remote detail tables.
- Nested materialized views must have a join or aggregate.
- Materialized join views and materialized aggregate views with a GROUP BY clause cannot select from an index-organized table.

Restrictions on Fast Refresh on Materialized Views with Joins Only

Defining queries for materialized views with joins only and no aggregates have the following restrictions on fast refresh:

- All restrictions from "General Restrictions on Fast Refresh" on page 9-20.
- They cannot have GROUP BY clauses or aggregates.
- Rowids of all the tables in the FROM list must appear in the SELECT list of the query.
- Materialized view logs must exist with rowids for all the base tables in the FROM list of the query.
- You cannot create a fast refreshable materialized view from multiple tables with simple joins that include an object type column in the SELECT statement.

Also, the refresh method you choose will not be optimally efficient if:

- The defining query uses an outer join that behaves like an inner join. If the defining query contains such a join, consider rewriting the defining query to contain an inner join.
- The SELECT list of the materialized view contains expressions on columns from multiple tables.

Restrictions on Fast Refresh on Materialized Views with Aggregates

Defining queries for materialized views with aggregates or joins have the following restrictions on fast refresh:

- All restrictions from "General Restrictions on Fast Refresh" on page 9-20.

Fast refresh is supported for both ON COMMIT and ON DEMAND materialized views, however the following restrictions apply:

- All tables in the materialized view must have materialized view logs, and the materialized view logs must:
  - Contain all columns from the table referenced in the materialized view. However, none of these columns in the base table can be encrypted.
  - Specify with ROWID and INCLUDING NEW VALUES.
  - Specify the SEQUENCE clause if the table is expected to have a mix of inserts/direct-loads, deletes, and updates.

- Only SUM, COUNT, AVG, STDDEV, VARIANCE, MIN and MAX are supported for fast refresh.
- COUNT(*) must be specified.
- Aggregate functions must occur only as the outermost part of the expression. That is, aggregates such as AVG(AVG(x)) or AVG(x)+AVG(x) are not allowed.
- For each aggregate such as AVG(expr), the corresponding COUNT(expr) must be present. Oracle recommends that SUM(expr) be specified. See Table 9–2 on page 9-11 for further details.
- If VARIANCE(expr) or STDDEV(expr) is specified, COUNT(expr) and SUM(expr) must be specified. Oracle recommends that SUM(expr *expr) be specified. See Table 9–2 on page 9-11 for further details.
- The SELECT column in the defining query cannot be a complex expression with columns from multiple base tables. A possible workaround to this is to use a nested materialized view.
- The SELECT list must contain all GROUP BY columns.
- If you use a CHAR data type in the filter columns of a materialized view log, the character sets of the master site and the materialized view must be the same.
- If the materialized view has one of the following, then fast refresh is supported only on conventional DML inserts and direct loads.
  - Materialized views with MIN or MAX aggregates
  - Materialized views which have SUM(expr) but no COUNT(expr)
  - Materialized views without COUNT(*)

  Such a materialized view is called an insert-only materialized view.
- A materialized view with MAX or MIN is fast refreshable after delete or mixed DML statements if it does not have a WHERE clause.
- Materialized views with named views or subqueries in the FROM clause can be fast refreshed provided the views can be completely merged. For information on which views will merge, refer to the *Oracle Database Performance Tuning Guide*.
- If there are no outer joins, you may have arbitrary selections and joins in the WHERE clause.
- Materialized aggregate views with outer joins are fast refreshable after conventional DML and direct loads, provided only the outer table has been modified. Also, unique constraints must exist on the join columns of the inner join table. If there are outer joins, all the joins must be connected by ANDs and must use the equality (=) operator.
- For materialized views with CUBE, ROLLUP, grouping sets, or concatenation of them, the following restrictions apply:
  - The SELECT list should contain grouping distinguisher that can either be a GROUPING_ID function on all GROUP BY expressions or GROUPING functions one for each GROUP BY expression. For example, if the GROUP BY clause of the materialized view is "GROUP BY CUBE(a, b)", then the SELECT list should contain either "GROUPING_ID(a, b)" or "GROUPING(a) AND GROUPING(b)" for the materialized view to be fast refreshable.
  - GROUP BY should not result in any duplicate groupings. For example, "GROUP BY a, ROLLUP(a, b)" is not fast refreshable because it results in duplicate groupings "(a), (a, b), AND (a)".

Restrictions on Fast Refresh on Materialized Views with UNION ALL

Materialized views with the UNION ALL set operator support the REFRESH FAST option if the following conditions are satisfied:

- The defining query must have the UNION ALL operator at the top level.

  The UNION ALL operator cannot be embedded inside a subquery, with one exception: The UNION ALL can be in a subquery in the FROM clause provided the defining query is of the form SELECT * FROM (view or subquery with UNION ALL) as in the following example:

```
CREATE VIEW view_with_unionall AS
(SELECT c.rowid crid, c.cust_id, 2 umarker
 FROM customers c WHERE c.cust_last_name = 'Smith'
 UNION ALL
 SELECT c.rowid crid, c.cust_id, 3 umarker
 FROM customers c WHERE c.cust_last_name = 'Jones');

CREATE MATERIALIZED VIEW unionall_inside_view_mv
REFRESH FAST ON DEMAND AS
SELECT * FROM view_with_unionall;
```

  Note that the view view_with_unionall satisfies the requirements for fast refresh.

- Each query block in the UNION ALL query must satisfy the requirements of a fast refreshable materialized view with aggregates or a fast refreshable materialized view with joins.

  The appropriate materialized view logs must be created on the tables as required for the corresponding type of fast refreshable materialized view.

  Note that the Oracle Database also allows the special case of a single table materialized view with joins only provided the ROWID column has been included in the SELECT list and in the materialized view log. This is shown in the defining query of the view view_with_unionall.

- The SELECT list of each query must include a UNION ALL marker, and the UNION ALL column must have a distinct constant numeric or string value in each UNION ALL branch. Further, the marker column must appear in the same ordinal position in the SELECT list of each query block. See "UNION ALL Marker" on page 19-46 for more information regarding UNION ALL markers.

- Some features such as outer joins, insert-only aggregate materialized view queries and remote tables are not supported for materialized views with UNION ALL. Note, however, that materialized views used in replication, which do not contain joins or aggregates, can be fast refreshed when UNION ALL or remote tables are used.

- The compatibility initialization parameter must be set to 9.2.0 or higher to create a fast refreshable materialized view with UNION ALL.

Achieving Refresh Goals

In addition to the EXPLAIN_MVIEW procedure, which is discussed throughout this chapter, you can use the DBMS_ADVISOR.TUNE_MVIEW procedure to optimize a CREATE MATERIALIZED VIEW statement to achieve REFRESH FAST and ENABLE QUERY REWRITE goals.

Refreshing Nested Materialized Views

A nested materialized view is considered to be fresh as long as its data is synchronized with the data in its detail tables, even if some of its detail tables could be stale materialized views.

You can refresh nested materialized views in two ways: DBMS_MVIEW.REFRESH with the nested flag set to TRUE and REFRESH_DEPENDENT with the nested flag set to TRUE on the base tables. If you use DBMS_MVIEW.REFRESH, the entire materialized view chain is refreshed and the coverage starting from the specified materialized view in top-down fashion. That is, the specified materialized view and all its child materialized views in the dependency hierarchy are refreshed in order. With DBMS_MVIEW.REFRESH_DEPENDENT, the entire chain is refreshed from the bottom up. That is, all the parent materialized views in the dependency hierarchy starting from the specified table are refreshed in order.

Example 9–7 Example of Refreshing a Nested Materialized View

The following statement shows an example of refreshing a nested materialized view:

```
DBMS_MVIEW.REFRESH('SALES_MV,COST_MV', nested => TRUE);
```

This statement will first refresh all child materialized views of sales_mv and cost_mv based on the dependency analysis and then refresh the two specified materialized views.

You can query the STALE_SINCE column in the *_MVIEWS views to find out when a materialized view became stale.

ORDER BY Clause

An ORDER BY clause is allowed in the CREATE MATERIALIZED VIEW statement. It is used only during the initial creation of the materialized view. It is not used during a full refresh or a fast refresh.

To improve the performance of queries against large materialized views, store the rows in the materialized view in the order specified in the ORDER BY clause. This initial ordering provides physical clustering of the data. If indexes are built on the columns by which the materialized view is ordered, accessing the rows of the materialized view using the index often reduces the time for disk I/O due to the physical clustering.

The ORDER BY clause is not considered part of the materialized view definition. As a result, there is no difference in the manner in which Oracle Database detects the various types of materialized views (for example, materialized join views with no aggregates). For the same reason, query rewrite is not affected by the ORDER BY clause. This feature is similar to the CREATE TABLE ... ORDER BY capability.

Materialized View Logs

Materialized view logs are required if you want to use fast refresh, with the exception of PCT refresh. That is, if a detail table supports PCT for a materialized view, the materialized view log on that detail table is not required in order to do fast refresh on that materialized view. As a general rule, though, you should create materialized view logs if you want to use fast refresh. Materialized view logs are defined using a CREATE MATERIALIZED VIEW LOG statement on the base table that is to be changed. They are not created on the materialized view unless there is another materialized view on top of that materialized view, which is the case with nested materialized views. For fast refresh of materialized views, the definition of the materialized view logs must

normally specify the ROWID clause. In addition, for aggregate materialized views, it must also contain every column in the table referenced in the materialized view, the INCLUDING NEW VALUES clause and the SEQUENCE clause. You can typically achieve better fast refresh performance of local materialized views containing aggregates or joins by using a WITH COMMIT SCN clause.

An example of a materialized view log is shown as follows where one is created on the table sales:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
(prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

Alternatively, you could create a commit SCN-based materialized view log as follows:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
 (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold),
COMMIT SCN INCLUDING NEW VALUES;
```

Oracle recommends that the keyword SEQUENCE be included in your materialized view log statement unless you are sure that you will never perform a mixed DML operation (a combination of INSERT, UPDATE, or DELETE operations on multiple tables). The SEQUENCE column is required in the materialized view log to support fast refresh with a combination of INSERT, UPDATE, or DELETE statements on multiple tables. You can, however, add the SEQUENCE number to the materialized view log after it has been created.

The boundary of a mixed DML operation is determined by whether the materialized view is ON COMMIT or ON DEMAND.

- For ON COMMIT, the mixed DML statements occur within the same transaction because the refresh of the materialized view will occur upon commit of this transaction.
- For ON DEMAND, the mixed DML statements occur between refreshes. The following example of a materialized view log illustrates where one is created on the table sales that includes the SEQUENCE keyword:

  ```
  CREATE MATERIALIZED VIEW LOG ON sales WITH SEQUENCE, ROWID
  (prod_id, cust_id, time_id, channel_id, promo_id,
   quantity_sold, amount_sold) INCLUDING NEW VALUES;
  ```

Using the FORCE Option with Materialized View Logs

If you specify FORCE and any items specified with the ADD clause have already been specified for the materialized view log, Oracle does not return an error, but silently ignores the existing elements and adds to the materialized view log any items that do not already exist in the log. For example, if you used a filter column such as cust_id and this column already existed, Oracle Database ignores the redundancy and does not return an error.

Materialized View Log Purging

Purging materialized view logs can be done during the materialized view refresh process or deferred until later, thus improving refresh performance time. You can choose different options for when the purge will occur, using a PURGE clause, as in the following:

```
CREATE MATERIALIZED VIEW LOG ON sales
PURGE START WITH sysdate NEXT sysdate+1
WITH ROWID
```

```
 (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold)
INCLUDING NEW VALUES;
```

You can also query USER_MVIEW_LOGS for purge information, as in the following:

```
SELECT PURGE_DEFERRED, PURGE_INTERVAL, LAST_PURGE_DATE, LAST_PURGE_STATUS
FROM USER_MVIEW_LOGS
WHERE LOG_OWNER "SH" AND MASTER = 'SALES';
```

In addition to setting the purge when creating a materialized view log, you can also modify an existing materialized view log by issuing a statement resembling the following:

```
ALTER MATERIALIZED VIEW LOG ON sales PURGE IMMEDIATE;
```

> **See Also:** *Oracle Database SQL Language Reference* for more information regarding materialized view log syntax

Using Oracle Enterprise Manager

A materialized view can also be created using Enterprise Manager by selecting the materialized view object type. There is no difference in the information required if this approach is used.

Using Materialized Views with NLS Parameters

When using certain materialized views, you must ensure that your NLS parameters are the same as when you created the materialized view. Materialized views with this restriction are as follows:

- Expressions that may return different values, depending on NLS parameter settings. For example, (`date > "01/02/03"`) or (`rate <= "2.150"`) are NLS parameter dependent expressions.
- Equijoins where one side of the join is character data. The result of this equijoin depends on collation and this can change on a session basis, giving an incorrect result in the case of query rewrite or an inconsistent materialized view after a refresh operation.
- Expressions that generate internal conversion to character data in the `SELECT` list of a materialized view, or inside an aggregate of a materialized aggregate view. This restriction does not apply to expressions that involve only numeric data, for example, `a+b` where `a` and `b` are numeric fields.

Adding Comments to Materialized Views

You can add a comment to a materialized view. For example, the following statement adds a comment to data dictionary views for the existing materialized view:

```
COMMENT ON MATERIALIZED VIEW sales_mv IS 'sales materialized view';
```

To view the comment after the preceding statement execution, the user can query the catalog views, {USER, DBA} ALL_MVIEW_COMMENTS. For example, consider the following example:

```
SELECT MVIEW_NAME, COMMENTS
FROM USER_MVIEW_COMMENTS WHERE MVIEW_NAME = 'SALES_MV';
```

The output will resemble the following:

```
MVIEW_NAME                     COMMENTS
-----------      ----------------------
SALES_MV         sales materialized view
```

Note: If the compatibility is set to 10.0.1 or higher, COMMENT ON TABLE will not be allowed for the materialized view container table. The following error message will be thrown if it is issued.

```
ORA-12098: cannot comment on the materialized view.
```

In the case of a prebuilt table, if it has an existing comment, the comment will be inherited by the materialized view after it has been created. The existing comment will be prefixed with '(from table)'. For example, table sales_summary was created to contain sales summary information. An existing comment 'Sales summary data' was associated with the table. A materialized view of the same name is created to use the prebuilt table as its container table. After the materialized view creation, the comment becomes '(from table) Sales summary data'.

However, if the prebuilt table, sales_summary, does not have any comment, the following comment is added: 'Sales summary data'. Then, if we drop the materialized view, the comment will be passed to the prebuilt table with the comment: '(from materialized view) Sales summary data'.

Registering Existing Materialized Views

Some data warehouses have implemented materialized views in ordinary user tables. Although this solution provides the performance benefits of materialized views, it does not:

- Provide query rewrite to all SQL applications.
- Enable materialized views defined in one application to be transparently accessed in another application.
- Generally support fast parallel or fast materialized view refresh.

Because of these limitations, and because existing materialized views can be extremely large and expensive to rebuild, you should register your existing materialized view tables whenever possible. You can register a user-defined materialized view with the CREATE MATERIALIZED VIEW ... ON PREBUILT TABLE statement. Once registered, the materialized view can be used for query rewrites or maintained by one of the refresh methods, or both.

The contents of the table must reflect the materialization of the defining query at the time you register it as a materialized view, and each column in the defining query must correspond to a column in the table that has a matching datatype. However, you can specify WITH REDUCED PRECISION to allow the precision of columns in the defining query to be different from that of the table columns.

The table and the materialized view must have the same name, but the table retains its identity as a table and can contain columns that are not referenced in the defining query of the materialized view. These extra columns are known as unmanaged columns. If rows are inserted during a refresh operation, each unmanaged column of the row is set to its default value. Therefore, the unmanaged columns cannot have NOT NULL constraints unless they also have default values.

Materialized views based on prebuilt tables are eligible for selection by query rewrite provided the parameter QUERY_REWRITE_INTEGRITY is set to STALE_TOLERATED or TRUSTED. See Chapter 18, "Basic Query Rewrite" for details about integrity levels.

When you drop a materialized view that was created on a prebuilt table, the table still exists—only the materialized view is dropped.

The following example illustrates the two steps required to register a user-defined table. First, the table is created, then the materialized view is defined using exactly the same name as the table. This materialized view `sum_sales_tab_mv` is eligible for use in query rewrite.

```
CREATE TABLE sum_sales_tab
PCTFREE 0  TABLESPACE demo
STORAGE (INITIAL 8M) AS
SELECT s.prod_id, SUM(amount_sold) AS dollar_sales,
       SUM(quantity_sold) AS unit_sales
FROM sales s GROUP BY s.prod_id;

CREATE MATERIALIZED VIEW sum_sales_tab_mv
ON PREBUILT TABLE WITHOUT REDUCED PRECISION
ENABLE QUERY REWRITE AS
SELECT s.prod_id, SUM(amount_sold) AS dollar_sales,
    SUM(quantity_sold) AS unit_sales
FROM sales s GROUP BY s.prod_id;
```

You could have compressed this table to save space. See "Storage And Table Compression" on page 9-16 for details regarding table compression.

In some cases, user-defined materialized views are refreshed on a schedule that is longer than the update cycle. For example, a monthly materialized view might be updated only at the end of each month, and the materialized view values always refer to complete time periods. Reports written directly against these materialized views implicitly select only data that is not in the current (incomplete) time period. If a user-defined materialized view already contains a time dimension:

- It should be registered and then fast refreshed each update cycle.
- You can create a view that selects the complete time period of interest.
- The reports should be modified to refer to the view instead of referring directly to the user-defined materialized view.

If the user-defined materialized view does not contain a time dimension, then:

- Create a new materialized view that does include the time dimension (if possible).
- The view should aggregate over the time column in the new materialized view.

Choosing Indexes for Materialized Views

The two most common operations on a materialized view are query execution and fast refresh, and each operation has different performance requirements. Query execution might need to access any subset of the materialized view key columns, and might need to join and aggregate over a subset of those columns. Consequently, query execution usually performs best if a single-column bitmap index is defined on each materialized view key column.

In the case of materialized views containing only joins using fast refresh, Oracle recommends that indexes be created on the columns that contain the rowids to improve the performance of the refresh operation.

If a materialized view using aggregates is fast refreshable, then an index appropriate for the fast refresh procedure is created unless `USING NO INDEX` is specified in the `CREATE MATERIALIZED VIEW` statement.

If the materialized view is partitioned, then, after doing a partition maintenance operation on the materialized view, the indexes become unusable, and they need to be rebuilt for fast refresh to work.

See *Oracle Database Performance Tuning Guide* for information on using the SQL Access Advisor to determine what indexes are appropriate for your materialized view.

Dropping Materialized Views

Use the `DROP MATERIALIZED VIEW` statement to drop a materialized view. For example, consider the following statement:

```
DROP MATERIALIZED VIEW sales_sum_mv;
```

This statement drops the materialized view `sales_sum_mv`. If the materialized view was prebuilt on a table, then the table is not dropped, but it can no longer be maintained with the refresh mechanism or used by query rewrite. Alternatively, you can drop a materialized view using Oracle Enterprise Manager.

Analyzing Materialized View Capabilities

You can use the `DBMS_MVIEW.EXPLAIN_MVIEW` procedure to learn what is possible with a materialized view or potential materialized view. In particular, this procedure enables you to determine:

- If a materialized view is fast refreshable
- What types of query rewrite you can perform with this materialized view
- Whether PCT refresh is possible

Using this procedure is straightforward. You simply call `DBMS_MVIEW.EXPLAIN_MVIEW`, passing in as a single parameter the schema and materialized view name for an existing materialized view. Alternatively, you can specify the `SELECT` string for a potential materialized view or the complete `CREATE MATERIALIZED VIEW` statement. The materialized view or potential materialized view is then analyzed and the results are written into either a table called `MV_CAPABILITIES_TABLE`, which is the default, or to an array called `MSG_ARRAY`.

Note that you must run the `utlxmv.sql` script prior to calling `EXPLAIN_MVIEW` except when you are placing the results in `MSG_ARRAY`. The script is found in the `admin` directory. It is to create the `MV_CAPABILITIES_TABLE` in the current schema. An explanation of the various capabilities is in Table 9–7 on page 9-32, and all the possible messages are listed in Table 9–8 on page 9-34.

Using the DBMS_MVIEW.EXPLAIN_MVIEW Procedure

The `EXPLAIN_MVIEW` procedure has the following parameters:

- `stmt_id`

  An optional parameter. A client-supplied unique identifier to associate output rows with specific invocations of `EXPLAIN_MVIEW`.

- `mv`

  The name of an existing materialized view or the query definition or the entire `CREATE MATERIALIZED VIEW` statement of a potential materialized view you want to analyze.

- `msg-array`

The PL/SQL VARRAY that receives the output.

EXPLAIN_MVIEW analyzes the specified materialized view in terms of its refresh and rewrite capabilities and inserts its results (in the form of multiple rows) into MV_CAPABILITIES_TABLE or MSG_ARRAY.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for further information about the DBMS_MVIEW package

DBMS_MVIEW.EXPLAIN_MVIEW Declarations

The following PL/SQL declarations that are made for you in the DBMS_MVIEW package show the order and datatypes of these parameters for explaining an existing materialized view and a potential materialized view with output to a table and to a VARRAY.

Explain an existing or potential materialized view with output to MV_CAPABILITIES_TABLE:

```
DBMS_MVIEW.EXPLAIN_MVIEW (mv           IN VARCHAR2,
                          stmt_id IN VARCHAR2:= NULL);
```

Explain an existing or potential materialized view with output to a VARRAY:

```
DBMS_MVIEW.EXPLAIN_MVIEW (mv          IN VARCHAR2,
                          msg_array   OUT SYS.ExplainMVArrayType);
```

Using MV_CAPABILITIES_TABLE

One of the simplest ways to use DBMS_MVIEW.EXPLAIN_MVIEW is with the MV_CAPABILITIES_TABLE, which has the following structure:

```
CREATE TABLE MV_CAPABILITIES_TABLE
(STATEMENT_ID      VARCHAR(30),   -- Client-supplied unique statement identifier
 MVOWNER           VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
 MVNAME            VARCHAR(30),   -- NULL for SELECT based EXPLAIN_MVIEW
 CAPABILITY_NAME   VARCHAR(30),   -- A descriptive name of the particular
                                  -- capability:
                                  -- REWRITE
                                  --   Can do at least full text match
                                  --   rewrite
                                  -- REWRITE_PARTIAL_TEXT_MATCH
                                  --   Can do at leat full and partial
                                  --   text match rewrite
                                  -- REWRITE_GENERAL
                                  --   Can do all forms of rewrite
                                  -- REFRESH
                                  --   Can do at least complete refresh
                                  -- REFRESH_FROM_LOG_AFTER_INSERT
                                  --   Can do fast refresh from an mv log
                                  --   or change capture table at least
                                  --   when update operations are
                                  --   restricted to INSERT
                                  -- REFRESH_FROM_LOG_AFTER_ANY
                                  --   can do fast refresh from an mv log
                                  --   or change capture table after any
                                  --   combination of updates
                                  -- PCT
                                  --   Can do Enhanced Update Tracking on
                                  --   the table named in the RELATED_NAME
                                  --   column.  EUT is needed for fast
                                  --   refresh after partitioned
```

```
                                -- maintenance operations on the table
                                -- named in the RELATED_NAME column
                                -- and to do non-stale tolerated
                                -- rewrite when the mv is partially
                                -- stale with respect to the table
                                -- named in the RELATED_NAME column.
                                -- EUT can also sometimes enable fast
                                -- refresh of updates to the table
                                -- named in the RELATED_NAME column
                                -- when fast refresh from an mv log
                                -- or change capture table is not
                                -- possible.
                                -- See Table 9-7
 POSSIBLE          CHARACTER(1), -- T = capability is possible
                                -- F = capability is not possible
 RELATED_TEXT      VARCHAR(2000), -- Owner.table.column, alias name, and so on
                                -- related to this message. The specific
                                -- meaning of this column depends on the
                                -- NSGNO column. See the documentation for
                                -- DBMS_MVIEW.EXPLAIN_MVIEW() for details.
 RELATED_NUM       NUMBER,       -- When there is a numeric value
                                -- associated with a row, it goes here.
 MSGNO             INTEGER,      -- When available, QSM message # explaining
                                -- why disabled or more details when
                                -- enabled.
 MSGTXT            VARCHAR(2000), -- Text associated with MSGNO.
 SEQ               NUMBER);      -- Useful in ORDER BY clause when
                                -- selecting from this table.
```

You can use the `utlxmv.sql` script found in the `admin` directory to create `MV_CAPABILITIES_TABLE`.

Example 9–8 DBMS_MVIEW.EXPLAIN_MVIEW

First, create the materialized view. Alternatively, you can use `EXPLAIN_MVIEW` on a potential materialized view using its `SELECT` statement or the complete `CREATE MATERIALIZED VIEW` statement.

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc,  SUM(s.amount_sold) AS dollars
FROM sales s,  times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Then, you invoke `EXPLAIN_MVIEW` with the materialized view to explain. You need to use the `SEQ` column in an `ORDER BY` clause so the rows will display in a logical order. If a capability is not possible, `N` will appear in the `P` column and an explanation in the `MSGTXT` column. If a capability is not possible for multiple reasons, a row is displayed for each reason.

```
EXECUTE DBMS_MVIEW.EXPLAIN_MVIEW ('SH.CAL_MONTH_SALES_MV');

SELECT capability_name,  possible, SUBSTR(related_text,1,8)
  AS rel_text, SUBSTR(msgtxt,1,60) AS msgtxt
FROM MV_CAPABILITIES_TABLE
ORDER BY seq;
```

```
CAPABILITY_NAME                P    REL_TEXT     MSGTXT
---------------                -    --------     ------
PCT                            N
REFRESH_COMPLETE               Y
REFRESH_FAST                   N
REWRITE                        Y
PCT_TABLE                      N    SALES        no partition key or PMARKER in select list
PCT_TABLE                      N    TIMES        relation is not a partitioned table
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.TIMES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have new values
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log must have ROWID
REFRESH_FAST_AFTER_INSERT      N    SH.SALES     mv log does not have all necessary columns
REFRESH_FAST_AFTER_ONETAB_DML  N    DOLLARS      SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ONETAB_DML  N                 see the reason why
                                                 REFRESH_FAST_AFTER_INSERT is disabled
REFRESH_FAST_AFTER_ONETAB_DML  N                 COUNT(*) is not present in the select list
REFRESH_FAST_AFTER_ONETAB_DML  N                 SUM(expr) without COUNT(expr)
REFRESH_FAST_AFTER_ANY_DML     N                 see the reason why
                                                 REFRESH_FAST_AFTER_ONETAB_DML is disabled
REFRESH_FAST_AFTER_ANY_DML     N    SH.TIMES     mv log must have sequence
REFRESH_FAST_AFTER_ANY_DML     N    SH.SALES     mv log must have sequence
REFRESH_PCT                    N                 PCT is not possible on any of the detail
                                                 tables in the materialized view
REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y
REWRITE_PCT                    N                 PCT is not possible on any detail tables
```

See Also:

- Chapter 16, "Maintaining the Data Warehouse" for further details about PCT
- Chapter 19, "Advanced Query Rewrite" for further details about PCT

MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details

Table 9–7 lists explanations for values in the CAPABILITY_NAME column.

Table 9–7 CAPABILITY_NAME Column Details

CAPABILITY_NAME	Description
PCT	If this capability is possible, Partition Change Tracking (PCT) is possible on at least one detail relation. If this capability is not possible, PCT is not possible with any detail relation referenced by the materialized view.
REFRESH_COMPLETE	If this capability is possible, complete refresh of the materialized view is possible.
REFRESH_FAST	If this capability is possible, fast refresh is possible at least under certain circumstances.
REWRITE	If this capability is possible, at least full text match query rewrite is possible. If this capability is not possible, no form of query rewrite is possible.

Table 9–7 (Cont.) CAPABILITY_NAME Column Details

CAPABILITY_NAME	Description
PCT_TABLE	If this capability is possible, it is possible with respect to a particular partitioned table in the top level FROM list. When possible, PCT applies to the partitioned table named in the RELATED_TEXT column. PCT is needed to support fast fresh after partition maintenance operations on the table named in the RELATED_TEXT column. PCT may also support fast refresh with regard to updates to the table named in the RELATED_TEXT column when fast refresh from a materialized view log is not possible. PCT is also needed to support query rewrite in the presence of partial staleness of the materialized view with regard to the table named in the RELATED_TEXT column. When disabled, PCT does not apply to the table named in the RELATED_TEXT column. In this case, fast refresh is not possible after partition maintenance operations on the table named in the RELATED_TEXT column. In addition, PCT-based refresh of updates to the table named in the RELATED_TEXT column is not possible. Finally, query rewrite cannot be supported in the presence of partial staleness of the materialized view with regard to the table named in the RELATED_TEXT column.
PCT_TABLE_ REWRITE	If this capability is possible, it is possible with respect to a particular partitioned table in the top level FROM list. When possible, PCT applies to the partitioned table named in the RELATED_TEXT column. This capability is needed to support query rewrite against this materialized view in partial stale state with regard to the table named in the RELATED_TEXT column. When disabled, query rewrite cannot be supported if this materialized view is in partial stale state with regard to the table named in the RELATED_TEXT column.
REFRESH_FAST_ AFTER_INSERT	If this capability is possible, fast refresh from a materialized view log is possible at least in the case where the updates are restricted to INSERT operations; complete refresh is also possible. If this capability is not possible, no form of fast refresh from a materialized view log is possible.
REFRESH_FAST_ AFTER_ONETAB_DML	If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation, provided all update operations are performed on a single table. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations are performed on multiple tables.
REFRESH_FAST_ AFTER_ANY_DML	If this capability is possible, fast refresh from a materialized view log is possible regardless of the type of update operation or the number of tables updated. If this capability is not possible, fast refresh from a materialized view log may not be possible when the update operations (other than INSERT) affect multiple tables.
REFRESH_FAST_PCT	If this capability is possible, fast refresh using PCT is possible. Generally, this means that refresh is possible after partition maintenance operations on those detail tables where PCT is indicated as possible.
REWRITE_FULL_ TEXT_MATCH	If this capability is possible, full text match query rewrite is possible. If this capability is not possible, full text match query rewrite is not possible.
REWRITE_PARTIAL_ TEXT_MATCH	If this capability is possible, at least full and partial text match query rewrite are possible. If this capability is not possible, at least partial text match query rewrite and general query rewrite are not possible.
REWRITE_GENERAL	If this capability is possible, all query rewrite capabilities are possible, including general query rewrite and full and partial text match query rewrite. If this capability is not possible, at least general query rewrite is not possible.
REWRITE_PCT	If this capability is possible, query rewrite can use a partially stale materialized view even in QUERY_REWRITE_INTEGRITY = ENFORCED or TRUSTED modes. When this capability is not possible, query rewrite can use a partially stale materialized view only in QUERY_ REWRITE_INTEGRITY = STALE_TOLERATED mode.

MV_CAPABILITIES_TABLE Column Details

Table 9–8 lists the semantics for RELATED_TEXT and RELATED_NUM columns.

Table 9–8 MV_CAPABILITIES_TABLE Column Details

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
NULL	NULL		For PCT capability only: [*owner*.]*name* of the table upon which PCT is enabled
2066	This statement resulted in an Oracle error	Oracle error number that occurred	
2067	No partition key or PMARKER or join dependent expression in SELECT list		[*owner*.]*name* of relation for which PCT is not supported
2068	Relation is not partitioned		[*owner*.]*name* of relation for which PCT is not supported
2069	PCT not supported with multicolumn partition key		[*owner*.]*name* of relation for which PCT is not supported
2070	PCT not supported with this type of partitioning		[*owner*.]*name* of relation for which PCT is not supported
2071	Internal error: undefined PCT failure code	The unrecognized numeric PCT failure code	[*owner*.]*name* of relation for which PCT is not supported
2072	Requirements not satisfied for fast refresh of nested materialized view		
2077	Materialized view log is newer than last full refresh		[*owner*.]*table_name* of table upon which the materialized view log is needed
2078	Materialized view log must have new values		[*owner*.]*table_name* of table upon which the materialized view log is needed
2079	Materialized view log must have ROWID		[*owner*.]*table_name* of table upon which the materialized view log is needed
2080	Materialized view log must have primary key		[*owner*.]*table_name* of table upon which the materialized view log is needed
2081	Materialized view log does not have all necessary columns		[*owner*.]*table_name* of table upon which the materialized view log is needed
2082	Problem with materialized view log		[*owner*.]*table_name* of table upon which the materialized view log is needed
2099	Materialized view references a remote table or view in the FROM list	Offset from the SELECT keyword to the table or view in question	[*owner*.]*name* of the table or view in question
2126	Multiple master sites		Name of the first different node, or NULL if the first different node is local
2129	Join or filter condition(s) are complex		[*owner*.]*name* of the table involved with the join or filter condition (or NULL when not available)
2130	Expression not supported for fast refresh	Offset from the SELECT keyword to the expression in question	The alias name in the SELECT list of the expression in question

Table 9–8 (Cont.) MV_CAPABILITIES_TABLE Column Details

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
2150	SELECT lists must be identical across the UNION operator	Offset from the SELECT keyword to the first different select item in the SELECT list	The alias name of the first different select item in the SELECT list
2182	PCT is enabled through a join dependency		[*owner*.]*name* of relation for which PCT_TABLE_REWRITE is not enabled
2183	Expression to enable PCT not in PARTITION BY of analytic function or model	The unrecognized numeric PCT failure code	[*owner*.]*name* of relation for which PCT is not enabled
2184	Expression to enable PCT cannot be rolled up		[*owner*.]*name* of relation for which PCT is not enabled
2185	No partition key or PMARKER in the SELECT list		[*owner*.]*name* of relation for which PCT_TABLE_REWRITE is not enabled
2186	GROUP OUTER JOIN is present		
2187	Materialized view on external table		

10

Advanced Materialized Views

This chapter discusses advanced topics in using materialized views. It contains the following topics:

Partitioning and Materialized Views

Because of the large volume of data held in a data warehouse, partitioning is an extremely useful option when designing a database. Partitioning the fact tables improves scalability, simplifies system administration, and makes it possible to define local indexes that can be efficiently rebuilt. Partitioning the fact tables also improves the opportunity of fast refreshing the materialized view because this may enable Partition Change Tracking (PCT) refresh on the materialized view. Partitioning a materialized view also has benefits for refresh, because the refresh procedure can then use parallel DML in more scenarios and PCT-based refresh can use truncate partition to efficiently maintain the materialized view. See *Oracle Database VLDB and Partitioning Guide* for further details about partitioning.

Partition Change Tracking

It is possible and advantageous to track freshness to a finer grain than the entire materialized view. The ability to identify which rows in a materialized view are affected by a certain detail table partition, is known as Partition Change Tracking. When one or more of the detail tables are partitioned, it may be possible to identify the specific rows in the materialized view that correspond to a modified detail partition(s); those rows become stale when a partition is modified while all other rows remain fresh.

You can use PCT to identify which materialized view rows correspond to a particular partition. PCT is also used to support fast refresh after partition maintenance operations on detail tables. For instance, if a detail table partition is truncated or dropped, the affected rows in the materialized view are identified and deleted.

Identifying which materialized view rows are fresh or stale, rather than considering the entire materialized view as stale, allows query rewrite to use those rows that are

fresh while in QUERY_REWRITE_INTEGRITY = ENFORCED or TRUSTED modes. Several views, such as DBA_MVIEW_DETAIL_PARTITION, detail which partitions are stale or fresh. Oracle does not rewrite against partial stale materialized views if partition change tracking on the changed table is enabled by the presence of join dependent expression in the materialized view. See "Join Dependent Expression" on page 10-3 for more information.

To support PCT, a materialized view must satisfy the following requirements:

- At least one of the detail tables referenced by the materialized view must be partitioned.
- Partitioned tables must use either range, list or composite partitioning.
- The top level partition key must consist of only a single column.
- The materialized view must contain either the partition key column or a partition marker or ROWID or join dependent expression of the detail table. See *Oracle Database PL/SQL Packages and Types Reference* for details regarding the DBMS_MVIEW.PMARKER function.
- If you use a GROUP BY clause, the partition key column or the partition marker or ROWID or join dependent expression must be present in the GROUP BY clause.
- If you use an analytic window function or the MODEL clause, the partition key column or the partition marker or ROWID or join dependent expression must be present in their respective PARTITION BY subclauses.
- Data modifications can only occur on the partitioned table. If PCT refresh is being done for a table which has join dependent expression in the materialized view, then data modifications should not have occurred in any of the join dependent tables.
- The COMPATIBILITY initialization parameter must be a minimum of 9.0.0.0.0.
- PCT is not supported for a materialized view that refers to views, remote tables, or outer joins.

Partition Key

Partition change tracking requires sufficient information in the materialized view to be able to correlate a detail row in the source partitioned detail table to the corresponding materialized view row. This can be accomplished by including the detail table partition key columns in the SELECT list and, if GROUP BY is used, in the GROUP BY list.

Consider an example of a materialized view storing daily customer sales. The following example uses the sh sample schema and the three detail tables sales, products, and times to create the materialized view. sales table is partitioned by time_id column and products is partitioned by the prod_id column. times is not a partitioned table.

Example 10–1 Partition Key

The following is an example:

```
CREATE MATERIALIZED VIEW LOG ON SALES WITH ROWID
   (prod_id, time_id, quantity_sold, amount_sold) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON PRODUCTS WITH ROWID
   (prod_id, prod_name, prod_desc) INCLUDING NEW VALUES;
CREATE MATERIALIZED VIEW LOG ON TIMES WITH ROWID
   (time_id, calendar_month_name, calendar_year) INCLUDING NEW VALUES;
```

```
CREATE MATERIALIZED VIEW cust_dly_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.time_id, p.prod_id, p.prod_name, COUNT(*),
         SUM(s.quantity_sold), SUM(s.amount_sold),
         COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY s.time_id, p.prod_id, p.prod_name;
```

For `cust_dly_sales_mv`, PCT is enabled on both the `sales` table and `products` table because their respective partitioning key columns `time_id` and `prod_id` are in the materialized view.

Join Dependent Expression

An expression consisting of columns from tables directly or indirectly joined through equijoins to the partitioned detail table on the partitioning key and which is either a dimensional attribute or a dimension hierarchical parent of the joining key is called a join dependent expression. The set of tables in the path to detail table are called join dependent tables. Consider the following:

```
SELECT s.time_id, t.calendar_month_name
FROM sales s, times t WHERE s.time_id = t.time_id;
```

In this query, `times` table is a join dependent table since it is joined to `sales` table on the partitioning key column `time_id`. Moreover, `calendar_month_name` is a dimension hierarchical attribute of `times.time_id`, because `calendar_month_name` is an attribute of `times.mon_id` and `times.mon_id` is a dimension hierarchical parent of `times.time_id`. Hence, the expression `calendar_month_name` from `times` tables is a join dependent expression. Let's consider another example:

```
SELECT s.time_id, y.calendar_year_name
FROM sales s, times_d d, times_m m, times_y y
WHERE s.time_id = d.time_id AND d.day_id = m.day_id AND m.mon_id = y.mon_id;
```

Here, `times` table is denormalized into `times_d`, `times_m` and `times_y` tables. The expression `calendar_year_name` from `times_y` table is a join dependent expression and the tables `times_d`, `times_m` and `times_y` are join dependent tables. This is because `times_y` table is joined indirectly through `times_m` and `times_d` tables to sales table on its partitioning key column `time_id`.

This lets users create materialized views containing aggregates on some level higher than the partitioning key of the detail table. Consider the following example of materialized view storing monthly customer sales.

Example 10–2 Join Dependent Expression

Assuming the presence of materialized view logs defined earlier, the materialized view can be created using the following DDL:

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD DEFERRED REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_name, p.prod_id, p.prod_name, COUNT(*),
    SUM(s.quantity_sold), SUM(s.amount_sold),
    COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
```

```
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, p.prod_id, p.prod_name;
```

Here, you can correlate a detail table row to its corresponding materialized view row using the join dependent table `times` and the relationship that `times.calendar_month_name` is a dimensional attribute determined by `times.time_id`. This enables partition change tracking on `sales` table. In addition to this, PCT is enabled on products table because of presence of its partitioning key column `prod_id` in the materialized view.

Partition Marker

The `DBMS_MVIEW.PMARKER` function is designed to significantly reduce the cardinality of the materialized view (see Example 10–3 for an example). The function returns a partition identifier that uniquely identifies the partition for a specified row within a specified partition table. Therefore, the `DBMS_MVIEW.PMARKER` function is used instead of the partition key column in the `SELECT` and `GROUP BY` clauses.

Unlike the general case of a PL/SQL function in a materialized view, use of the `DBMS_MVIEW.PMARKER` does not prevent rewrite with that materialized view even when the rewrite mode is `QUERY_REWRITE_INTEGRITY = ENFORCED`.

As an example of using the `PMARKER` function, consider calculating a typical number, such as revenue generated by a product category during a given year. If there were 1000 different products sold each month, it would result in 12,000 rows in the materialized view.

Example 10–3 Partition Marker

Consider an example of a materialized view storing the yearly sales revenue for each product category. With approximately hundreds of different products in each product category, including the partitioning key column `prod_id` of the `products` table in the materialized view would substantially increase the cardinality. Instead, this materialized view uses the `DBMS_MVIEW.PMARKER` function, which increases the cardinality of materialized view by a factor of the number of partitions in the `products` table.

```
CREATE MATERIALIZED VIEW prod_yr_sales_mv
BUILD DEFERRED
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(p.rowid), p.prod_category, t.calendar_year, COUNT(*),
       SUM(s.amount_sold), SUM(s.quantity_sold),
       COUNT(s.amount_sold), COUNT(s.quantity_sold)
FROM   sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY  DBMS_MVIEW.PMARKER (p.rowid), p.prod_category, t.calendar_year;
```

`prod_yr_sales_mv` includes the `DBMS_MVIEW.PMARKER` function on the `products` table in its `SELECT` list. This enables partition change tracking on `products` table with significantly less cardinality impact than grouping by the partition key column `prod_id`. In this example, the desired level of aggregation for the `prod_yr_sales_mv` is to group by `products.prod_category`. Using the `DBMS_MVIEW.PMARKER` function, the materialized view cardinality is increased only by a factor of the number of partitions in the `products` table. This would generally be significantly less than the cardinality impact of including the partition key columns.

Note that partition change tracking is enabled on `sales` table because of presence of join dependent expression `calendar_year` in the `SELECT` list.

Partial Rewrite

A subsequent INSERT statement adds a new row to the sales_part3 partition of table sales. At this point, because cust_dly_sales_mv has PCT available on table sales using a partition key, Oracle can identify the stale rows in the materialized view cust_dly_sales_mv corresponding to sales_part3 partition (The other rows are unchanged in their freshness state). Query rewrite cannot identify the fresh portion of materialized views cust_mth_sales_mv and prod_yr_sales_mv because PCT is available on table sales using join dependent expressions. Query rewrite can determine the fresh portion of a materialized view on changes to a detail table only if PCT is available on the detail table using a partition key or partition marker.

Partitioning a Materialized View

Partitioning a materialized view involves defining the materialized view with the standard Oracle partitioning clauses, as illustrated in the following example. This statement creates a materialized view called part_sales_mv, which uses three partitions, can be fast refreshed, and is eligible for query rewrite:

```
CREATE MATERIALIZED VIEW part_sales_mv
PARALLEL PARTITION BY RANGE (time_id)
(PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf1,
 PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf2,
 PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf3)
BUILD DEFERRED
REFRESH FAST
ENABLE QUERY REWRITE AS
SELECT s.cust_id, s.time_id,
  SUM(s.amount_sold) AS sum_dol_sales, SUM(s.quantity_sold) AS sum_unit_sales
  FROM sales s GROUP BY s.time_id, s.cust_id;
```

Partitioning a Prebuilt Table

Alternatively, a materialized view can be registered to a partitioned prebuilt table as illustrated in the following example:

```
CREATE TABLE part_sales_tab_mv(time_id, cust_id, sum_dollar_sales, sum_unit_sale)
PARALLEL PARTITION BY RANGE (time_id)
(PARTITION month1
      VALUES LESS THAN (TO_DATE('31-12-1998', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf1,
 PARTITION month2
      VALUES LESS THAN (TO_DATE('31-12-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
```

```
      TABLESPACE sf2,
PARTITION month3
      VALUES LESS THAN (TO_DATE('31-12-2000', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M)
      TABLESPACE sf3)  AS
SELECT s.time_id, s.cust_id, SUM(s.amount_sold) AS sum_dollar_sales,
  SUM(s.quantity_sold) AS sum_unit_sales
FROM sales s GROUP BY s.time_id, s.cust_id;

CREATE MATERIALIZED VIEW part_sales_tab_mv
ON PREBUILT TABLE
ENABLE QUERY REWRITE AS
SELECT s.time_id,  s.cust_id, SUM(s.amount_sold) AS sum_dollar_sales,
     SUM(s.quantity_sold) AS sum_unit_sales
FROM sales s GROUP BY s.time_id, s.cust_id;
```

In this example, the table `part_sales_tab_mv` has been partitioned over three months and then the materialized view was registered to use the prebuilt table. This materialized view is eligible for query rewrite because the `ENABLE QUERY REWRITE` clause has been included.

Benefits of Partitioning a Materialized View

When a materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table, it is more efficient to use a `TRUNCATE PARTITION` statement to remove one or more partitions of the materialized view during refresh and then repopulate the partition with new data. Oracle Database uses this variant of fast refresh (called PCT refresh) with partition truncation if the following conditions are satisfied in addition to other conditions described in "Partition Change Tracking" on page 10-1.

- The materialized view is partitioned on the partitioning key column or join dependent expressions of the detail table.
- If PCT is enabled using either the partitioning key column or join expressions, the materialized view should be range or list partitioned.
- PCT refresh is nonatomic.

Rolling Materialized Views

When a data warehouse or data mart contains a time dimension, it is often desirable to archive the oldest information and then reuse the storage for new information. This is called the rolling window scenario. If the fact tables or materialized views include a time dimension and are horizontally partitioned by the time attribute, then management of rolling materialized views can be reduced to a few fast partition maintenance operations provided the unit of data that is rolled out equals, or is at least aligned with, the range partitions.

If you plan to have rolling materialized views in your data warehouse, you should determine how frequently you plan to perform partition maintenance operations, and you should plan to partition fact tables and materialized views to reduce the amount of system administration overhead required when old data is aged out. An additional consideration is that you might want to use data compression on your infrequently updated partitions.

You are not restricted to using range partitions. For example, a composite partition using both a time value and a key value could result in a good partition solution for your data.

See Chapter 16, "Maintaining the Data Warehouse" for further details regarding `CONSIDER FRESH` and for details regarding compression.

Materialized Views in Analytic Processing Environments

This section discusses the concepts used by analytic SQL and how relational databases can handle these types of queries. It also illustrates the best approach for creating materialized views using a common scenario.

Cubes

While data warehouse environments typically view data in the form of a star schema, for analytical SQL queries, data is held in the form of a hierarchical cube. A hierarchical cube includes the data aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. It includes the typical set of aggregations needed for business intelligence queries.

Example 10–4 Hierarchical Cube

Consider a sales data set with two dimensions, each of which has a 4-level hierarchy:

- Time, which contains (all times), year, quarter, and month.
- Product, which contains (all products), division, brand, and item.

This means there are 16 aggregate groups in the hierarchical cube. This is because the four levels of time are multiplied by four levels of product to produce the cube. Table 10–1 shows the four levels of each dimension.

Table 10–1 ROLLUP By Time and Product

ROLLUP By Time	ROLLUP By Product
year, quarter, month	division, brand, item
year, quarter	division, brand
year	division
all times	all products

Note that as you increase the number of dimensions and levels, the number of groups to calculate increases dramatically. This example involves 16 groups, but if you were to add just two more dimensions with the same number of levels, you would have 4 x 4 x 4 x 4 = 256 different groups. Also, consider that a similar increase in groups occurs if you have multiple hierarchies in your dimensions. For example, the time dimension might have an additional hierarchy of fiscal month rolling up to fiscal quarter and then fiscal year. Handling the explosion of groups has historically been the major challenge in data storage for online analytical processing systems.

Typical online analytical queries **slice and dice** different parts of the cube comparing aggregations from one level to aggregation from another level. For instance, a query might find sales of the grocery division for the month of January, 2002 and compare them with total sales of the grocery division for all of 2001.

Benefits of Partitioning Materialized Views

Materialized views with multiple aggregate groups give their best performance for refresh and query rewrite when partitioned appropriately.

PCT refresh in a rolling window scenario requires partitioning at the top level on some level from the time dimension. And, partition pruning for queries rewritten against this materialized view requires partitioning on GROUPING_ID column. Hence, the most effective partitioning scheme for these materialized views is to use composite partitioning (range-list on (time, GROUPING_ID) columns). By partitioning the materialized views this way, you enable:

- PCT refresh, thereby improving refresh performance.
- Partition pruning: only relevant aggregate groups are accessed, thereby greatly reducing the query processing cost.

If you do not want to use PCT refresh, you can just partition by list on GROUPING_ID column.

Compressing Materialized Views

You should consider data compression when using highly redundant data, such as tables with many foreign keys. In particular, materialized views created with the ROLLUP clause are likely candidates. See *Oracle Database SQL Language Reference* for data compression syntax and restrictions and "Storage And Table Compression" on page 9-16 for details regarding compression.

Materialized Views with Set Operators

Oracle Database provides support for materialized views whose defining query involves set operators. Materialized views with set operators can now be created enabled for query rewrite. You can refresh the materialized view using either ON COMMIT or ON DEMAND refresh.

Fast refresh is supported if the defining query has the UNION ALL operator at the top level and each query block in the UNION ALL, meets the requirements of a materialized view with aggregates or materialized view with joins only. Further, the materialized view must include a constant column (known as a UNION ALL marker) that has a distinct value in each query block, which, in the following example, is columns 1 marker and 2 marker.

See "Restrictions on Fast Refresh on Materialized Views with UNION ALL" on page 9-23 for detailed restrictions on fast refresh for materialized views with UNION ALL.

Examples of Materialized Views Using UNION ALL

The following examples illustrate creation of fast refreshable materialized views involving UNION ALL.

Example 10–5 Materialized View Using UNION ALL with Two Join Views

To create a UNION ALL materialized view with two join views, the materialized view logs must have the rowid column and, in the following example, the UNION ALL marker is the columns, 1 marker and 2 marker.

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID;
CREATE MATERIALIZED VIEW LOG ON customers WITH ROWID;

CREATE MATERIALIZED VIEW unionall_sales_cust_joins_mv
REFRESH FAST ON COMMIT
ENABLE QUERY REWRITE AS
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 1 marker
FROM sales s, customers c
```

```
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Smith')
UNION ALL
(SELECT c.rowid crid, s.rowid srid, c.cust_id, s.amount_sold, 2 marker
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND c.cust_last_name = 'Brown');
```

Example 10–6 Materialized View Using UNION ALL with Joins and Aggregates

The following example shows a UNION ALL of a materialized view with joins and a materialized view with aggregates. A couple of things can be noted in this example. Nulls or constants can be used to ensure that the data types of the corresponding SELECT list columns match. Also, the UNION ALL marker column can be a string literal, which is 'Year' umarker, 'Quarter' umarker, or 'Daily' umarker in the following example:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID, SEQUENCE
(amount_sold, time_id)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON times WITH ROWID, SEQUENCE
  (time_id, fiscal_year, fiscal_quarter_number, day_number_in_week)
INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW unionall_sales_mix_mv
REFRESH FAST ON DEMAND AS
(SELECT 'Year' umarker, NULL, NULL, t.fiscal_year,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
 FROM sales s, times t
 WHERE s.time_id = t.time_id
 GROUP BY t.fiscal_year)
UNION ALL
(SELECT 'Quarter' umarker, NULL, NULL, t.fiscal_quarter_number,
        SUM(s.amount_sold) amt, COUNT(s.amount_sold), COUNT(*)
FROM sales s, times t
WHERE s.time_id = t.time_id and t.fiscal_year = 2001
GROUP BY t.fiscal_quarter_number)
UNION ALL
(SELECT 'Daily' umarker, s.rowid rid, t.rowid rid2, t.day_number_in_week,
        s.amount_sold amt, 1, 1
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.time_id between '01-Jan-01' AND '01-Dec-31');
```

Materialized Views and Models

Models, which provide array-based computations in SQL, can be used in materialized views. Because the MODEL clause calculations can be expensive, you may want to use two separate materialized views: one for the model calculations and one for the SELECT ... GROUP BY query. For example, instead of using one, long materialized view, you could create the following materialized views:

```
CREATE MATERIALIZED VIEW my_groupby_mv
REFRESH FAST
ENABLE QUERY REWRITE AS
SELECT country_name country, prod_name prod, calendar_year year,
  SUM(amount_sold) sale, COUNT(amount_sold) cnt, COUNT(*) cntstr
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
```

```
      customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year;

CREATE MATERIALIZED VIEW my_model_mv
ENABLE QUERY REWRITE AS
SELECT country, prod, year, sale, cnt
FROM my_groupby_mv
MODEL PARTITION BY(country) DIMENSION BY(prod, year)
   MEASURES(sale s) IGNORE NAV
(s['Shorts', 2000] =  0.2 * AVG(s)[CV(), year BETWEEN 1996 AND 1999],
s['Kids Pajama', 2000] = 0.5 * AVG(s)[CV(), year BETWEEN 1995 AND 1999],
s['Boys Pajama', 2000] = 0.6 * AVG(s)[CV(), year BETWEEN 1994 AND 1999],
...
<hundreds of other update rules>);
```

By using two materialized views, you can incrementally maintain the materialized view `my_groupby_mv`. The materialized view `my_model_mv` is on a much smaller data set because it is built on `my_groupby_mv` and can be maintained by a complete refresh.

Materialized views with models can use complete refresh or PCT refresh only, and are available for partial text query rewrite only.

> **See Also:** Chapter 23, "SQL for Modeling" for further details about model calculations

Invalidating Materialized Views

Dependencies related to materialized views are automatically maintained to ensure correct operation. When a materialized view is created, the materialized view depends on the detail tables referenced in its definition. Any DML operation, such as an `INSERT`, or `DELETE`, `UPDATE`, or DDL operation on any dependency in the materialized view will cause it to become invalid. To revalidate a materialized view, use the `ALTER MATERIALIZED VIEW COMPILE` statement.

A materialized view is automatically revalidated when it is referenced. In many cases, the materialized view will be successfully and transparently revalidated. However, if a column has been dropped in a table referenced by a materialized view or the owner of the materialized view did not have one of the query rewrite privileges and that privilege has now been granted to the owner, you should use the following statement to revalidate the materialized view:

```
ALTER MATERIALIZED VIEW mview_name COMPILE;
```

The state of a materialized view can be checked by querying the data dictionary views `USER_MVIEWS` or `ALL_MVIEWS`. The column `STALENESS` will show one of the values `FRESH`, `STALE`, `UNUSABLE`, `UNKNOWN`, `UNDEFINED`, or `NEEDS_COMPILE` to indicate whether the materialized view can be used. The state is maintained automatically. However, if the staleness of a materialized view is marked as `NEEDS_COMPILE`, you could issue an `ALTER MATERIALIZED VIEW ... COMPILE` statement to validate the materialized view and get the correct staleness state. If the state of a materialized view is `UNUSABLE`, you must perform a complete refresh to bring the materialized view back to the `FRESH` state. If the materialized view is based on a prebuilt table that you never refresh, you must drop and re-create the materialized view. The staleness of remote materialized views is not tracked. Thus, if you use remote materialized views for rewrite, they are considered to be trusted.

Security Issues with Materialized Views

To create a materialized view in your own schema, you must have the CREATE MATERIALIZED VIEW privilege and the SELECT privilege to any tables referenced that are in another schema. To create a materialized view in another schema, you must have the CREATE ANY MATERIALIZED VIEW privilege and the owner of the materialized view needs SELECT privileges to the tables referenced if they are from another schema. Moreover, if you enable query rewrite on a materialized view that references tables outside your schema, you must have the GLOBAL QUERY REWRITE privilege or the QUERY REWRITE object privilege on each table outside your schema.

If the materialized view is on a prebuilt container, the creator, if different from the owner, must have SELECT WITH GRANT privilege on the container table.

If you continue to get a privilege error while trying to create a materialized view and you believe that all the required privileges have been granted, then the problem is most likely due to a privilege not being granted explicitly and trying to inherit the privilege from a role instead. The owner of the materialized view must have explicitly been granted SELECT access to the referenced tables if the tables are in a different schema.

If the materialized view is being created with ON COMMIT REFRESH specified, then the owner of the materialized view requires an additional privilege if any of the tables in the defining query are outside the owner's schema. In that case, the owner requires the ON COMMIT REFRESH system privilege or the ON COMMIT REFRESH object privilege on each table outside the owner's schema.

Querying Materialized Views with Virtual Private Database (VPD)

For all security concerns, a materialized view serves as a view that happens to be materialized when you are directly querying the materialized view. When creating a view or materialized view, the owner must have the necessary permissions to access the underlying base relations of the view or materialized view that they are creating. With these permissions, the owner can publish a view or materialized view that other users can access, assuming they have been granted access to the view or materialized view.

Using materialized views with Virtual Private Database is similar. When you create a materialized view, there must not be any VPD policies in effect against the base relations of the materialized view for the owner of the materialized view. However, the owner of the materialized view may establish a VPD policy on the new materialized view. Users who access the materialized view are subject to the VPD policy on the materialized view. However, they are not additionally subject to the VPD policies of the underlying base relations of the materialized view, since security processing of the underlying base relations is performed against the owner of the materialized view.

Using Query Rewrite with Virtual Private Database

When you access a materialized view using query rewrite, the materialized view serves as an access structure much like an index. As such, the security implications for materialized views accessed in this way are much the same as for indexes: all security checks are performed against the relations specified in the request query. The index or materialized view is used to speed the performance of accessing the data, not provide any additional security checks. Thus, the presence of the index or materialized view presents no additional security checking.

This holds true when you are accessing a materialized view using query rewrite in the presence of VPD. The request query is subject to any VPD policies that are present against the relations specified in the query. Query rewrite may rewrite the query to use a materialize view instead of accessing the detail relations, but only if it can guarantee to deliver exactly the same rows as if the rewrite had not occurred. Specifically, query rewrite must retain and respect any VPD policies against the relations specified in the request query. However, any VPD policies against the materialized view itself do not have effect when the materialized view is accessed using query rewrite. This is because the data is already protected by the VPD policies against the relations in the request query.

Restrictions with Materialized Views and Virtual Private Database

Query rewrite does not use its full and partial text match modes with request queries that include relations with active VPD policies, but it does use general rewrite methods. This is because VPD transparently transforms the request query to affect the VPD policy. If query rewrite were to perform a text match transformation against a request query with a VPD policy, the effect would be to negate the VPD policy.

In addition, when you create or refresh a materialized view, the owner of the materialized view must not have any active VPD policies in effect against the base relations of the materialized view, or an error is returned. The materialized view owner must either have no such VPD policies, or any such policy must return `NULL`. This is because VPD would transparently modify the defining query of the materialized view such that the set of rows contained by the materialized view would not match the set of rows indicated by the materialized view definition.

One way to work around this restriction yet still create a materialized view containing the desired VPD-specified subset of rows is to create the materialized view in a user account that has no active VPD policies against the detail relations of the materialized view. In addition, you can include a predicate in the `WHERE` clause of the materialized view that embodies the effect of the VPD policy. When query rewrite attempts to rewrite a request query that has that VPD policy, it matches up the VPD-generated predicate on the request query with the predicate you directly specify when you create the materialized view.

Altering Materialized Views

Six modifications can be made to a materialized view. You can:

- Change its refresh option (`FAST/FORCE/COMPLETE/NEVER`).
- Change its refresh mode (`ON COMMIT/ON DEMAND`).
- Recompile it.
- Enable or disable its use for query rewrite.
- Consider it fresh.
- Partition maintenance operations.

All other changes are achieved by dropping and then re-creating the materialized view.

The `COMPILE` clause of the `ALTER MATERIALIZED VIEW` statement can be used when the materialized view has been invalidated. This compile process is quick, and allows the materialized view to be used by query rewrite again.

See Also:

- *Oracle Database SQL Language Reference* for further information about the `ALTER MATERIALIZED VIEW` statement
- "Invalidating Materialized Views" on page 10-10

11

Dimensions

This chapter discusses using dimensions in a data warehouse: It contains the following topics:

What are Dimensions?

A **dimension** is a structure that categorizes data in order to enable users to answer business questions. Commonly used dimensions are customers, products, and time. For example, each sales channel of a clothing retailer might gather and store data regarding sales and reclamations of their Cloth assortment. The retail chain management can build a data warehouse to analyze the sales of its products across all stores over time and help answer questions such as:

- What is the effect of promoting one product on the sale of a related product that is not promoted?
- What are the sales of a product before and after a promotion?
- How does a promotion affect the various distribution channels?

The data in the retailer's data warehouse system has two important components: dimensions and facts. The dimensions are products, customers, promotions, channels, and time. One approach for identifying your dimensions is to review your reference tables, such as a product table that contains everything about a product, or a promotion table containing all information about promotions. The facts are sales (units sold) and profits. A data warehouse contains facts about the sales of each product at on a daily basis.

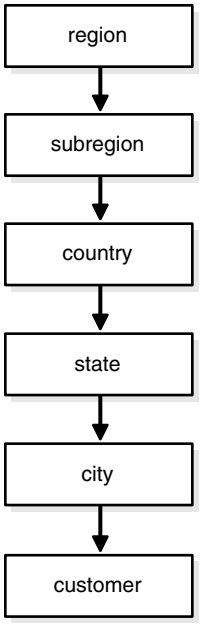
A typical relational implementation for such a data warehouse is a star schema. The fact information is stored in what is called a fact table, whereas the dimensional information is stored in dimension tables. In our example, each sales transaction record is uniquely defined as for each customer, for each product, for each sales channel, for each promotion, and for each day (time).

See Also: Chapter 20, "Schema Modeling Techniques" for further details

In Oracle Database, the dimensional information itself is stored in a dimension table. In addition, the database object dimension helps to organize and group dimensional information into hierarchies. This represents natural `1:n` relationships between columns or column groups (the levels of a hierarchy) that cannot be represented with constraint conditions. Going up a level in the hierarchy is called rolling up the data and going down a level in the hierarchy is called drilling down the data. In the retailer example:

- Within the `time` dimension, months roll up to quarters, quarters roll up to years, and years roll up to all years.
- Within the `product` dimension, products roll up to subcategories, subcategories roll up to categories, and categories roll up to all products.
- Within the `customer` dimension, customers roll up to `city`. Then cities roll up to `state`. Then states roll up to `country`. Then countries roll up to `subregion`. Finally, subregions roll up to `region`, as shown in Figure 11–1.

Figure 11–1 Sample Rollup for a Customer Dimension



Data analysis typically starts at higher levels in the dimensional hierarchy and gradually drills down if the situation warrants such analysis.

Dimensions do not have to be defined. However, if your application uses dimensional modeling, it is worth spending time creating them as it can yield significant benefits, because they help query rewrite perform more complex types of rewrite. Dimensions are also beneficial to certain types of materialized view refresh operations and with the SQL Access Advisor. They are only mandatory if you use the SQL Access Advisor (a GUI tool for materialized view and index management) without a workload to recommend which materialized views and indexes to create, drop, or retain.

See Also:

- Chapter 18, "Basic Query Rewrite" for further details regarding query rewrite
- *Oracle Database Performance Tuning Guide* for further details regarding the SQL Access Advisor

In spite of the benefits of dimensions, you must not create dimensions in any schema that does not fully satisfy the dimensional relationships described in this chapter. Incorrect results can be returned from queries otherwise.

Creating Dimensions

Before you can create a dimension object, the dimension tables must exist in the database possibly containing the dimension data. For example, if you create a customer dimension, one or more tables must exist that contain the city, state, and country information. In a star schema data warehouse, these dimension tables already exist. It is therefore a simple task to identify which ones will be used.

Now you can draw the hierarchies of a dimension as shown in Figure 11–1. For example, `city` is a child of `state` (because you can aggregate city-level data up to state), and `country`. This hierarchical information will be stored in the database object dimension.

In the case of normalized or partially normalized dimension representation (a dimension that is stored in more than one table), identify how these tables are joined. Note whether the joins between the dimension tables can guarantee that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, determine whether the child-side columns uniquely determine the parent-side (or attribute) columns. If you use constraints to represent these relationships, they can be enabled with the `NOVALIDATE` and `RELY` clauses if the relationships represented by the constraints are guaranteed by other means.

You may want the capability to skip `NULL` levels in a dimension. An example of this is with Puerto Rico. You may want Puerto Rico to be included within a region of North America, but not include it within the state category. If you want this capability, use the `SKIP WHEN NULL` clause. See the sample dimension later in this section for more information and *Oracle Database SQL Language Reference* for syntax and restrictions.

You create a dimension using either the `CREATE DIMENSION` statement or the Dimension Wizard in Oracle Enterprise Manager. Within the `CREATE DIMENSION` statement, use the `LEVEL` clause to identify the names of the dimension levels.

See Also: *Oracle Database SQL Language Reference* for a complete description of the `CREATE DIMENSION` statement

This customer dimension contains a single hierarchy with a geographical rollup, with arrows drawn from the child level to the parent level, as shown in Figure 11–1 on page 11-2.

Each arrow in this graph indicates that for any child there is one and only one parent. For example, each city must be contained in exactly one state and each state must be contained in exactly one country. States that belong to more than one country violate hierarchical integrity. Also, you must use the `SKIP WHEN NULL` clause if you want to include cities that do not belong to a state, such as Washington D.C. Hierarchical integrity is necessary for the correct operation of management functions for materialized views that include aggregates.

For example, you can declare a dimension products_dim, which contains levels product, subcategory, and category:

```
CREATE DIMENSION products_dim
       LEVEL product           IS (products.prod_id)
       LEVEL subcategory       IS (products.prod_subcategory)
       LEVEL category          IS (products.prod_category) ...
```

Each level in the dimension must correspond to one or more columns in a table in the database. Thus, level product is identified by the column prod_id in the products table and level subcategory is identified by a column called prod_subcategory in the same table.

In this example, the database tables are denormalized and all the columns exist in the same table. However, this is not a prerequisite for creating dimensions. "Using Normalized Dimension Tables" on page 11-8 shows how to create a dimension customers_dim that has a normalized schema design using the JOIN KEY clause.

The next step is to declare the relationship between the levels with the HIERARCHY statement and give that hierarchy a name. A hierarchical relationship is a functional dependency from one level of a hierarchy to the next level in the hierarchy. Using the level names defined previously, the CHILD OF relationship denotes that each child's level value is associated with one and only one parent level value. The following statement declares a hierarchy prod_rollup and defines the relationship between products, subcategory, and category:

```
HIERARCHY prod_rollup
 (product         CHILD OF
  subcategory     CHILD OF
  category)
```

In addition to the 1:n hierarchical relationships, dimensions also include 1:1 attribute relationships between the hierarchy levels and their dependent, determined dimension attributes. For example, the dimension times_dim, as defined in *Oracle Database Sample Schemas*, has columns fiscal_month_desc, fiscal_month_name, and days_in_fiscal_month. Their relationship is defined as follows:

```
LEVEL fis_month IS TIMES.FISCAL_MONTH_DESC
...
ATTRIBUTE fis_month DETERMINES
      (fiscal_month_name, days_in_fiscal_month)
```

The ATTRIBUTE ... DETERMINES clause relates fis_month to fiscal_month_name and days_in_fiscal_month. Note that this is a unidirectional determination. It is only guaranteed, that for a specific fiscal_month, for example, 1999-11, you will find exactly one matching values for fiscal_month_name, for example, November and days_in_fiscal_month, for example, 28. You cannot determine a specific fiscal_month_desc based on the fiscal_month_name, which is November for every fiscal year.

In this example, suppose a query were issued that queried by fiscal_month_name instead of fiscal_month_desc. Because this 1:1 relationship exists between the attribute and the level, an already aggregated materialized view containing fiscal_month_desc can be joined back to the dimension information and used to identify the data.

> **See Also:** Chapter 18, "Basic Query Rewrite" for further details of using dimensional information

A sample dimension definition follows:

```
CREATE DIMENSION products_dim
    LEVEL product          IS (products.prod_id)
    LEVEL subcategory      IS (products.prod_subcategory) [SKIP WHEN NULL]
    LEVEL category         IS (products.prod_category)
    HIERARCHY prod_rollup (
         product         CHILD OF
         subcategory     CHILD OF
         category)
    ATTRIBUTE product DETERMINES
     (products.prod_name, products.prod_desc,
      prod_weight_class, prod_unit_of_measure,
      prod_pack_size, prod_status, prod_list_price, prod_min_price)
    ATTRIBUTE subcategory DETERMINES
     (prod_subcategory, prod_subcategory_desc)
    ATTRIBUTE category DETERMINES
     (prod_category, prod_category_desc);
```

Alternatively, the *extended_attribute_clause* could have been used instead of the *attribute_clause*, as shown in the following example:

```
CREATE DIMENSION products_dim
    LEVEL product          IS (products.prod_id)
    LEVEL subcategory      IS (products.prod_subcategory)
    LEVEL category         IS (products.prod_category)
    HIERARCHY prod_rollup (
         product         CHILD OF
         subcategory     CHILD OF
         category
        )
    ATTRIBUTE product_info LEVEL product DETERMINES
     (products.prod_name, products.prod_desc,
      prod_weight_class, prod_unit_of_measure,
      prod_pack_size, prod_status, prod_list_price, prod_min_price)
    ATTRIBUTE subcategory DETERMINES
     (prod_subcategory, prod_subcategory_desc)
    ATTRIBUTE category DETERMINES
     (prod_category, prod_category_desc);
```

The design, creation, and maintenance of dimensions is part of the design, creation, and maintenance of your data warehouse schema. Once the dimension has been created, verify that it meets these requirements:

- There must be a `1:n` relationship between a parent and children. A parent can have one or more children, but a child can have only one parent.
- There must be a `1:1` attribute relationship between hierarchy levels and their dependent dimension attributes. For example, if there is a column `fiscal_month_desc`, then a possible attribute relationship would be `fiscal_month_desc` to `fiscal_month_name`. For skip `NULL` levels, if a row of the relation of a skip level has a `NULL` value for the level column, then that row must have a `NULL` value for the attribute-relationship column, too.
- If the columns of a parent level and child level are in different relations, then the connection between them also requires a `1:n` join relationship. Each row of the child table must join with one and only one row of the parent table unless you use the `SKIP WHEN NULL` clause. This relationship is stronger than referential integrity alone, because it requires that the child join key must be non-null, that referential integrity must be maintained from the child join key to the parent join key, and that the parent join key must be unique.

- You must ensure (using database constraints if necessary) that the columns of each hierarchy level are non-null unless you use the SKIP WHEN NULL clause and that hierarchical integrity is maintained.
- An optional join key is a join key that connects the immediate non-skip child (if such a level exists), CHILDLEV, of a skip level to the nearest non-skip ancestor (again, if such a level exists), ANCLEV, of the skip level in the hierarchy. Also, this joinkey is allowed only when CHILDLEV and ANCLEV are defined over different relations.
- The hierarchies of a dimension can overlap or be disconnected from each other. However, the columns of a hierarchy level cannot be associated with more than one dimension.
- Join relationships that form cycles in the dimension graph are not supported. For example, a hierarchy level cannot be joined to itself either directly or indirectly.

Note: The information stored with a dimension objects is only declarative. The previously discussed relationships are not enforced with the creation of a dimension object. You should validate any dimension definition with the DBMS_DIMENSION.VALIDATE_DIMENSION procedure, as discussed in "Validating Dimensions" on page 11-10.

Dropping and Creating Attributes with Columns

You can use the attribute clause in a CREATE DIMENSION statement to specify one or multiple columns that are uniquely determined by a hierarchy level.

If you use the *extended_attribute_clause* to create multiple columns determined by a hierarchy level, you can drop one attribute column without dropping them all. Alternatively, you can specify an attribute name for each attribute clause CREATE or ALTER DIMENSION statement so that an attribute name is specified for each attribute clause where multiple level-to-column relationships can be individually specified.

The following statement illustrates how you can drop a single column without dropping all columns:

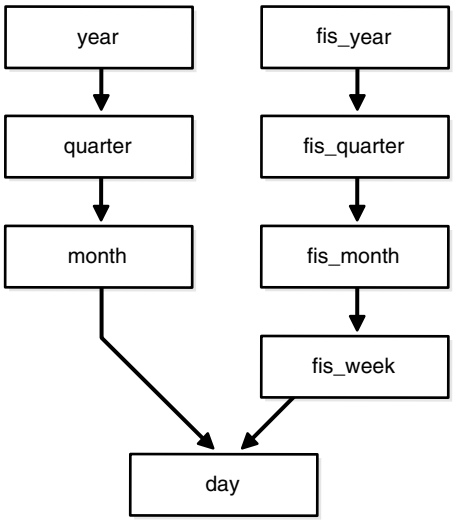
```
CREATE DIMENSION products_dim
LEVEL product         IS (products.prod_id)
LEVEL subcategory     IS (products.prod_subcategory)
LEVEL category        IS (products.prod_category)
HIERARCHY prod_rollup (
          product        CHILD OF
          subcategory    CHILD OF category)
ATTRIBUTE product DETERMINES
         (products.prod_name, products.prod_desc,
          prod_weight_class, prod_unit_of_measure,
          prod_pack_size,prod_status, prod_list_price, prod_min_price)
ATTRIBUTE subcategory_att DETERMINES
         (prod_subcategory, prod_subcategory_desc)
ATTRIBUTE category DETERMINES
         (prod_category, prod_category_desc);

ALTER DIMENSION products_dim
DROP ATTRIBUTE subcategory_att LEVEL subcategory COLUMN prod_subcategory;
```

See Also: *Oracle Database SQL Language Reference* for a complete description of the `CREATE DIMENSION` statement

Multiple Hierarchies

A single dimension definition can contain multiple hierarchies. Suppose our retailer wants to track the sales of certain items over time. The first step is to define the time dimension over which sales will be tracked. Figure 11–2 illustrates a dimension `times_dim` with two time hierarchies.

Figure 11–2 times_dim Dimension with Two Time Hierarchies



From the illustration, you can construct the hierarchy of the denormalized `time_dim` dimension's `CREATE DIMENSION` statement as follows. The complete `CREATE DIMENSION` statement as well as the `CREATE TABLE` statement are shown in *Oracle Database Sample Schemas*.

```
CREATE DIMENSION times_dim
   LEVEL day         IS times.time_id
   LEVEL month       IS times.calendar_month_desc
   LEVEL quarter     IS times.calendar_quarter_desc
   LEVEL year        IS times.calendar_year
   LEVEL fis_week    IS times.week_ending_day
   LEVEL fis_month   IS times.fiscal_month_desc
   LEVEL fis_quarter IS times.fiscal_quarter_desc
   LEVEL fis_year    IS times.fiscal_year
   HIERARCHY cal_rollup    (
             day     CHILD OF
             month   CHILD OF
             quarter CHILD OF
             year
   )
   HIERARCHY fis_rollup    (
             day         CHILD OF
             fis_week    CHILD OF
             fis_month   CHILD OF
             fis_quarter CHILD OF
             fis_year
   ) <attribute determination clauses>;
```

Using Normalized Dimension Tables

The tables used to define a dimension may be normalized or denormalized and the individual hierarchies can be normalized or denormalized. If the levels of a hierarchy come from the same table, it is called a fully denormalized hierarchy. For example, `cal_rollup` in the `times_dim` dimension is a denormalized hierarchy. If levels of a hierarchy come from different tables, such a hierarchy is either a fully or partially normalized hierarchy. This section shows how to define a normalized hierarchy.

Suppose the tracking of a customer's location is done by city, state, and country. This data is stored in the tables customers and countries. The customer dimension `customers_dim` is partially normalized because the data entities `cust_id` and `country_id` are taken from different tables. The clause `JOIN KEY` within the dimension definition specifies how to join together the levels in the hierarchy. The dimension statement is partially shown in the following. The complete `CREATE DIMENSION` statement as well as the `CREATE TABLE` statement are shown in *Oracle Database Sample Schemas*.

```
CREATE DIMENSION customers_dim
   LEVEL customer  IS (customers.cust_id)
   LEVEL city      IS (customers.cust_city)
   LEVEL state     IS (customers.cust_state_province)
   LEVEL country   IS (countries.country_id)
   LEVEL subregion IS (countries.country_subregion)
   LEVEL region IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer        CHILD OF
      city            CHILD OF
      state           CHILD OF
      country         CHILD OF
      subregion       CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country);
```

If you use the `SKIP WHEN NULL` clause, you can use the `JOIN KEY` clause to link levels that have a missing level in their hierarchy. For example, the following statement enables a state level that has been declared as `SKIP WHEN NULL` to join city and country:

```
JOIN KEY (city.country_id) REFERENCES country;
```

This ensures that the rows at customer and city levels can still be associated with the rows of country, subregion, and region levels.

Viewing Dimensions

Dimensions can be viewed through one of two methods:

- Using Oracle Enterprise Manager
- Using the DESCRIBE_DIMENSION Procedure

Using Oracle Enterprise Manager

All of the dimensions that exist in the data warehouse can be viewed using Oracle Enterprise Manager. Select the **Dimension** object from within the **Schema** icon to display all of the dimensions. Select a specific dimension to graphically display its hierarchy, levels, and any attributes that have been defined.

Using the DESCRIBE_DIMENSION Procedure

To view the definition of a dimension, use the DESCRIBE_DIMENSION procedure in the DBMS_DIMENSION package. For example, if a dimension is created in the sh sample schema with the following statements:

```
CREATE DIMENSION channels_dim
        LEVEL channel       IS (channels.channel_id)
        LEVEL channel_class IS (channels.channel_class)
        HIERARCHY channel_rollup (
                channel CHILD OF channel_class)
        ATTRIBUTE channel DETERMINES (channel_desc)
        ATTRIBUTE channel_class DETERMINES (channel_class);
```

Execute the DESCRIBE_DIMENSION procedure as follows:

```
SET SERVEROUTPUT ON FORMAT WRAPPED;  --to improve the display of info
EXECUTE DBMS_DIMENSION.DESCRIBE_DIMENSION('SH.CHANNELS_DIM');
```

You then see the following output results:

```
EXECUTE DBMS_DIMENSION.DESCRIBE_DIMENSION('SH.CHANNELS_DIM');
  DIMENSION SH.CHANNELS_DIM
    LEVEL CHANNEL IS SH.CHANNELS.CHANNEL_ID
    LEVEL CHANNEL_CLASS IS SH.CHANNELS.CHANNEL_CLASS

    HIERARCHY CHANNEL_ROLLUP (
             CHANNEL CHILD OF
             CHANNEL_CLASS)

    ATTRIBUTE CHANNEL LEVEL CHANNEL DETERMINES
SH.CHANNELS.CHANNEL_DESC
    ATTRIBUTE CHANNEL_CLASS LEVEL CHANNEL_CLASS DETERMINES
SH.CHANNELS.CHANNEL_CLASS
```

Using Dimensions with Constraints

Constraints play an important role with dimensions. Full referential integrity is sometimes enabled in data warehouses, but not always. This is because operational databases normally have full referential integrity and you can ensure that the data flowing into your data warehouse never violates the already established integrity rules.

It is recommended that constraints be enabled and, if validation time is a concern, then the NOVALIDATE clause should be used as follows:

```
ENABLE NOVALIDATE CONSTRAINT pk_time;
```

Primary and foreign keys should be implemented also. Referential integrity constraints and NOT NULL constraints on the fact tables provide information that query rewrite can use to extend the usefulness of materialized views.

In addition, you should use the RELY clause to inform query rewrite that it can rely upon the constraints being correct as follows:

```
ALTER TABLE time MODIFY CONSTRAINT pk_time RELY;
```

This information is also used for query rewrite. See Chapter 18, "Basic Query Rewrite" for more information.

If you use the SKIP WHEN NULL clause, at least one of the referenced level columns should not have NOT NULL constraints.

Validating Dimensions

The information of a dimension object is declarative only and not enforced by the database. If the relationships described by the dimensions are incorrect, incorrect results could occur. Therefore, you should verify the relationships specified by CREATE DIMENSION using the DBMS_DIMENSION.VALIDATE_DIMENSION procedure periodically.

This procedure is easy to use and has only four parameters:

- dimension: the owner and name.
- incremental: set to TRUE to check only the new rows for tables of this dimension.
- check_nulls: set to TRUE to verify that all columns that are not in the levels containing a SKIP WHEN NULL clause are not null.
- statement_id: a user-supplied unique identifier to identify the result of each run of the procedure.

The following example validates the dimension TIME_FN in the sh schema:

```
@utldim.sql
EXECUTE DBMS_DIMENSION.VALIDATE_DIMENSION ('SH.TIME_FN', FALSE, TRUE,
  'my first example');
```

Before running the VALIDATE_DIMENSION procedure, you need to create a local table, DIMENSION_EXCEPTIONS, by running the provided script utldim.sql. If the VALIDATE_DIMENSION procedure encounters any errors, they are placed in this table. Querying this table will identify the exceptions that were found. The following illustrates a sample:

```
SELECT * FROM dimension_exceptions
WHERE statement_id = 'my first example';

STATEMENT_ID       OWNER  TABLE_NAME  DIMENSION_NAME   RELATIONSHIP   BAD_ROWID
------------       -----  ----------  --------------   ------------   ---------
my first example   SH     MONTH       TIME_FN          FOREIGN KEY    AAAAuwAAJAAAARwAAA
```

However, rather than query this table, it may be better to query the rowid of the invalid row to retrieve the actual row that has violated the constraint. In this example, the dimension TIME_FN is checking a table called month. It has found a row that violates the constraints. Using the rowid, you can see exactly which row in the month table is causing the problem, as in the following:

```
SELECT * FROM month
WHERE rowid IN (SELECT bad_rowid
                FROM dimension_exceptions
                WHERE statement_id = 'my first example');

MONTH    QUARTER   FISCAL_QTR    YEAR    FULL_MONTH_NAME    MONTH_NUMB
------   -------   ----------    ----    ---------------    ----------
199903     19981        19981    1998               March             3
```

Altering Dimensions

You can modify a dimension using the ALTER DIMENSION statement. You can add or drop a level, hierarchy, or attribute from the dimension using this command.

Referring to the time dimension in Figure 11–2 on page 11-7, you can remove the attribute `fis_year`, drop the hierarchy `fis_rollup`, or remove the level `fiscal_year`. In addition, you can add a new level called `f_year` as in the following:

```
ALTER DIMENSION times_dim DROP ATTRIBUTE fis_year;
ALTER DIMENSION times_dim DROP HIERARCHY fis_rollup;
ALTER DIMENSION times_dim DROP LEVEL fis_year;
ALTER DIMENSION times_dim ADD LEVEL f_year IS times.fiscal_year;
```

If you used the *extended_attribute_clause* when creating the dimension, you can drop one attribute column without dropping all attribute columns. This is illustrated in "Dropping and Creating Attributes with Columns" on page 11-6, which shows the following statement:

```
ALTER DIMENSION product_dim
DROP ATTRIBUTE size LEVEL prod_type COLUMN Prod_TypeSize;
```

If you try to remove anything with further dependencies inside the dimension, Oracle Database rejects the altering of the dimension. A dimension becomes invalid if you change any schema object that the dimension is referencing. For example, if the table on which the dimension is defined is altered, the dimension becomes invalid.

You can modify a dimension by adding a level containing a `SKIP WHEN NULL` clause, as in the following statement:

```
ALTER DIMENSION times_dim
ADD LEVEL f_year IS times.fiscal_year SKIP WHEN NULL;
```

You cannot, however, modify a level that contains a `SKIP WHEN NULL` clause. Instead, you need to drop the level and re-create it.

To check the status of a dimension, view the contents of the column `invalid` in the `ALL_DIMENSIONS` data dictionary view. To revalidate the dimension, use the `COMPILE` option as follows:

```
ALTER DIMENSION times_dim COMPILE;
```

Dimensions can also be modified or deleted using Oracle Enterprise Manager.

Deleting Dimensions

A dimension is removed using the `DROP DIMENSION` statement. For example, you could issue the following statement:

```
DROP DIMENSION times_dim;
```

Part IV

Managing the Data Warehouse Environment

This section discusses the tasks necessary for managing a data warehouse.

It contains the following chapters:

12

Overview of Extraction, Transformation, and Loading

This chapter discusses the process of extracting, transporting, transforming, and loading data in a data warehousing environment. It includes the following topics:

Overview of ETL in Data Warehouses

You must load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the data warehouse. The challenge in data warehouse environments is to integrate, rearrange and consolidate large volumes of data over many systems, thereby providing a new unified information base for business intelligence.

The process of extracting data from source systems and bringing it into the data warehouse is commonly called **ETL**, which stands for extraction, transformation, and loading. Note that ETL refers to a broad process, and not three well-defined steps. The acronym ETL is perhaps too simplistic, because it omits the transportation phase and implies that each of the other phases of the process is distinct. Nevertheless, the entire process is known as ETL.

The methodology and tasks of ETL have been well known for many years, and are not necessarily unique to data warehouse environments: a wide variety of proprietary applications and database systems are the IT backbone of any enterprise. Data has to be shared between applications or systems, trying to integrate them, giving at least two applications the same picture of the world. This data sharing was mostly addressed by mechanisms similar to what we now call ETL.

ETL Basics in Data Warehousing

What happens during the ETL process? The following tasks are the main actions in the process.

Extraction of Data

During extraction, the desired data is identified and extracted from many different sources, including database systems and applications. Very often, it is not possible to identify the specific subset of interest, therefore more data than necessary has to be extracted, so the identification of the relevant data will be done at a later point in time. Depending on the source system's capabilities (for example, operating system

resources), some transformations may take place during this extraction process. The size of the extracted data varies from hundreds of kilobytes up to gigabytes, depending on the source system and the business situation. The same is true for the time delta between two (logically) identical extractions: the time span may vary between days/hours and minutes to near real-time. Web server log files, for example, can easily grow to hundreds of megabytes in a very short period.

Transportation of Data

After data is extracted, it has to be physically transported to the target system or to an intermediate system for further processing. Depending on the chosen way of transportation, some transformations can be done during this process, too. For example, a SQL statement which directly accesses a remote target through a gateway can concatenate two columns as part of the `SELECT` statement.

The emphasis in many of the examples in this section is scalability. Many long-time users of Oracle Database are experts in programming complex data transformation logic using PL/SQL. These chapters suggest alternatives for many such data manipulation operations, with a particular emphasis on implementations that take advantage of Oracle's new SQL functionality, especially for ETL and the parallel query infrastructure.

ETL Tools for Data Warehouses

Designing and maintaining the ETL process is often considered one of the most difficult and resource-intensive portions of a data warehouse project. Many data warehousing projects use ETL tools to manage this process. Oracle Warehouse Builder, for example, provides ETL capabilities and takes advantage of inherent database abilities. Other data warehouse builders create their own ETL tools and processes, either inside or outside the database.

Besides the support of extraction, transformation, and loading, there are some other tasks that are important for a successful ETL implementation as part of the daily operations of the data warehouse and its support for further enhancements. Besides the support for designing a data warehouse and the data flow, these tasks are typically addressed by ETL tools such as Oracle Warehouse Builder.

Oracle is not an ETL tool and does not provide a complete solution for ETL. However, Oracle does provide a rich set of capabilities that can be used by both ETL tools and customized ETL solutions. Oracle offers techniques for transporting data between Oracle databases, for transforming large volumes of data, and for quickly loading new data into a data warehouse.

Daily Operations in Data Warehouses

The successive loads and transformations must be scheduled and processed in a specific order. Depending on the success or failure of the operation or parts of it, the result must be tracked and subsequent, alternative processes might be started. The control of the progress as well as the definition of a business workflow of the operations are typically addressed by ETL tools such as Oracle Warehouse Builder.

Evolution of the Data Warehouse

As the data warehouse is a living IT system, sources and targets might change. Those changes must be maintained and tracked through the lifespan of the system without overwriting or deleting the old ETL process flow information. To build and keep a level of trust about the information in the warehouse, the process flow of each

individual record in the warehouse can be reconstructed at any point in time in the future in an ideal case.

13

Extraction in Data Warehouses

This chapter discusses extraction, which is the process of taking data from an operational system and moving it to your data warehouse or staging system. The chapter discusses:

Overview of Extraction in Data Warehouses

Extraction is the operation of extracting data from a source system for further use in a data warehouse environment. This is the first step of the ETL process. After the extraction, this data can be transformed and loaded into the data warehouse.

The source systems for a data warehouse are typically transaction processing applications. For example, one of the source systems for a sales analysis data warehouse might be an order entry system that records all of the current order activities.

Designing and creating the extraction process is often one of the most time-consuming tasks in the ETL process and, indeed, in the entire data warehousing process. The source systems might be very complex and poorly documented, and thus determining which data needs to be extracted can be difficult. The data has to be extracted normally not only once, but several times in a periodic manner to supply all changed data to the data warehouse and keep it up-to-date. Moreover, the source system typically cannot be modified, nor can its performance or availability be adjusted, to accommodate the needs of the data warehouse extraction process.

These are important considerations for extraction and ETL in general. This chapter, however, focuses on the technical considerations of having different kinds of sources and extraction methods. It assumes that the data warehouse team has already identified the data that will be extracted, and discusses common techniques used for extracting data from source databases.

Designing this process means making decisions about the following two main aspects:

- Which extraction method do I choose?

  This influences the source system, the transportation process, and the time needed for refreshing the warehouse.

- How do I provide the extracted data for further processing?

  This influences the transportation method, and the need for cleaning and transforming the data.

Introduction to Extraction Methods in Data Warehouses

The extraction method you should choose is highly dependent on the source system and also from the business needs in the target data warehouse environment. Very often, there is no possibility to add additional logic to the source systems to enhance an incremental extraction of data due to the performance or the increased workload of these systems. Sometimes even the customer is not allowed to add anything to an out-of-the-box application system.

Logical Extraction Methods

There are two types of logical extraction:

- Full Extraction
- Incremental Extraction

Full Extraction

The data is extracted completely from the source system. Because this extraction reflects all the data currently available on the source system, there's no need to keep track of changes to the data source since the last successful extraction. The source data will be provided as-is and no additional logical information (for example, timestamps) is necessary on the source site. An example for a full extraction may be an export file of a distinct table or a remote SQL statement scanning the complete source table.

Incremental Extraction

At a specific point in time, only the data that has changed since a well-defined event back in history is extracted. This event may be the last time of extraction or a more complex business event like the last booking day of a fiscal period. To identify this delta change there must be a possibility to identify all the changed information since this specific time event. This information can be either provided by the source data itself such as an application column, reflecting the last-changed timestamp or a change table where an appropriate additional mechanism keeps track of the changes besides the originating transactions. In most cases, using the latter method means adding extraction logic to the source system.

Many data warehouses do not use any change-capture techniques as part of the extraction process. Instead, entire tables from the source systems are extracted to the data warehouse or staging area, and these tables are compared with a previous extract from the source system to identify the changed data. This approach may not have significant impact on the source systems, but it clearly can place a considerable burden on the data warehouse processes, particularly if the data volumes are large.

Oracle's Change Data Capture (CDC) mechanism can extract and maintain such delta information. See Chapter 17, "Change Data Capture" for further details about the Change Data Capture framework.

Physical Extraction Methods

Depending on the chosen logical extraction method and the capabilities and restrictions on the source side, the extracted data can be physically extracted by two mechanisms. The data can either be extracted online from the source system or from an offline structure. Such an offline structure might already exist or it might be generated by an extraction routine.

There are the following methods of physical extraction:

Online Extraction

The data is extracted directly from the source system itself. The extraction process can connect directly to the source system to access the source tables themselves or to an intermediate system that stores the data in a preconfigured manner (for example, snapshot logs or change tables). Note that the intermediate system is not necessarily physically different from the source system.

With online extractions, you must consider whether the distributed transactions are using original source objects or prepared source objects.

Offline Extraction

The data is not extracted directly from the source system but is staged explicitly outside the original source system. The data already has an existing structure (for example, redo logs, archive logs or transportable tablespaces) or was created by an extraction routine.

You should consider the following structures:

- Flat files

  Data in a defined, generic format. Additional information about the source object is necessary for further processing.

- Dump files

  Oracle-specific format. Information about the containing objects may or may not be included, depending on the chosen utility.

- Redo and archive logs

  Information is in a special, additional dump file.

- Transportable tablespaces

  A powerful way to extract and move large volumes of data between Oracle databases. A more detailed example of using this feature to extract and transport data is provided in Chapter 14, "Transportation in Data Warehouses". Oracle recommends that you use transportable tablespaces whenever possible, because they can provide considerable advantages in performance and manageability over other extraction techniques.

  See *Oracle Database Utilities* for more information on using export/import.

Change Data Capture

An important consideration for extraction is incremental extraction, also called Change Data Capture. If a data warehouse extracts data from an operational system on a nightly basis, then the data warehouse requires only the data that has changed since the last extraction (that is, the data that has been modified in the past 24 hours). Change Data Capture is also the key-enabling technology for providing near real-time, or on-time, data warehousing.

When it is possible to efficiently identify and extract only the most recently changed data, the extraction process (and all downstream operations in the ETL process) can be much more efficient, because it must extract a much smaller volume of data. Unfortunately, for many source systems, identifying the recently modified data may

be difficult or intrusive to the operation of the system. Change Data Capture is typically the most challenging technical issue in data extraction.

Because change data capture is often desirable as part of the extraction process and it might not be possible to use the Change Data Capture mechanism, this section describes several techniques for implementing a self-developed change capture on Oracle Database source systems:

- Timestamps
- Partitioning
- Triggers

These techniques are based upon the characteristics of the source systems, or may require modifications to the source systems. Thus, each of these techniques must be carefully evaluated by the owners of the source system prior to implementation.

Each of these techniques can work in conjunction with the data extraction technique discussed previously. For example, timestamps can be used whether the data is being unloaded to a file or accessed through a distributed query. See Chapter 17, "Change Data Capture" for further details.

Timestamps

The tables in some operational systems have timestamp columns. The timestamp specifies the time and date that a given row was last modified. If the tables in an operational system have columns containing timestamps, then the latest data can easily be identified using the timestamp columns. For example, the following query might be useful for extracting today's data from an `orders` table:

```
SELECT * FROM orders
WHERE TRUNC(CAST(order_date AS date),'dd') =
  TO_DATE(SYSDATE,'dd-mon-yyyy');
```

If the timestamp information is not available in an operational source system, you are not always able to modify the system to include timestamps. Such modification would require, first, modifying the operational system's tables to include a new timestamp column and then creating a trigger to update the timestamp column following every operation that modifies a given row.

Partitioning

Some source systems might use range partitioning, such that the source tables are partitioned along a date key, which allows for easy identification of new data. For example, if you are extracting from an `orders` table, and the `orders` table is partitioned by week, then it is easy to identify the current week's data.

Triggers

Triggers can be created in operational systems to keep track of recently updated records. They can then be used in conjunction with timestamp columns to identify the exact time and date when a given row was last modified. You do this by creating a trigger on each source table that requires change data capture. Following each DML statement that is executed on the source table, this trigger updates the timestamp column with the current time. Thus, the timestamp column provides the exact time and date when a given row was last modified.

A similar internalized trigger-based technique is used for Oracle materialized view logs. These logs are used by materialized views to identify changed data, and these

logs are accessible to end users. However, the format of the materialized view logs is not documented and might change over time.

If you want to use a trigger-based mechanism, use synchronous change data capture. It is recommended that you use synchronous Change Data Capture for trigger based change capture, because CDC provides an externalized interface for accessing the change information and provides a framework for maintaining the distribution of this information to various clients.

Materialized view logs rely on triggers, but they provide an advantage in that the creation and maintenance of this change-data system is largely managed by the database.

However, Oracle recommends the usage of synchronous Change Data Capture for trigger-based change capture, since CDC provides an externalized interface for accessing the change information and provides a framework for maintaining the distribution of this information to various clients

Trigger-based techniques might affect performance on the source systems, and this impact should be carefully considered prior to implementation on a production source system.

Data Warehousing Extraction Examples

You can extract data in two ways:

- Extraction Using Data Files
- Extraction Through Distributed Operations

Extraction Using Data Files

Most database systems provide mechanisms for exporting or unloading data from the internal database format into flat files. Extracts from mainframe systems often use COBOL programs, but many databases, and third-party software vendors, provide export or unload utilities.

Data extraction does not necessarily mean that entire database structures are unloaded in flat files. In many cases, it may be appropriate to unload entire database tables or objects. In other cases, it may be more appropriate to unload only a subset of a given table such as the changes on the source system since the last extraction or the results of joining multiple tables together. Different extraction techniques vary in their capabilities to support these two scenarios.

When the source system is an Oracle database, several alternatives are available for extracting data into files:

- Extracting into Flat Files Using SQL*Plus
- Extracting into Flat Files Using OCI or Pro*C Programs
- Exporting into Export Files Using the Export Utility
- Extracting into Export Files Using External Tables

Extracting into Flat Files Using SQL*Plus

The most basic technique for extracting data is to execute a SQL query in SQL*Plus and direct the output of the query to a file. For example, to extract a flat file, `country_city.log`, with the pipe sign as delimiter between column values,

containing a list of the cities in the US in the tables `countries` and `customers`, the following SQL script could be run:

```
SET echo off SET pagesize 0 SPOOL country_city.log
SELECT distinct t1.country_name ||'|'|| t2.cust_city
FROM countries t1, customers t2 WHERE t1.country_id = t2.country_id
AND t1.country_name= 'United States of America';
SPOOL off
```

The exact format of the output file can be specified using SQL*Plus system variables.

This extraction technique offers the advantage of storing the result in a customized format. Note that, using the external table data pump unload facility, you can also extract the result of an arbitrary SQL operation. The example previously extracts the results of a join.

This extraction technique can be parallelized by initiating multiple, concurrent SQL*Plus sessions, each session running a separate query representing a different portion of the data to be extracted. For example, suppose that you wish to extract data from an `orders` table, and that the `orders` table has been range partitioned by month, with partitions `orders_jan1998`, `orders_feb1998`, and so on. To extract a single year of data from the `orders` table, you could initiate 12 concurrent SQL*Plus sessions, each extracting a single partition. The SQL script for one such session could be:

```
SPOOL order_jan.dat
SELECT * FROM orders PARTITION (orders_jan1998);
SPOOL OFF
```

These 12 SQL*Plus processes would concurrently spool data to 12 separate files. You can then concatenate them if necessary (using operating system utilities) following the extraction. If you are planning to use SQL*Loader for loading into the target, these 12 files can be used as is for a parallel load with 12 SQL*Loader sessions. See Chapter 14, "Transportation in Data Warehouses" for an example.

Even if the `orders` table is not partitioned, it is still possible to parallelize the extraction either based on logical or physical criteria. The logical method is based on logical ranges of column values, for example:

```
SELECT ... WHERE order_date
BETWEEN TO_DATE('01-JAN-99') AND TO_DATE('31-JAN-99');
```

The physical method is based on a range of values. By viewing the data dictionary, it is possible to identify the Oracle Database data blocks that make up the `orders` table. Using this information, you could then derive a set of rowid-range queries for extracting data from the `orders` table:

```
SELECT * FROM orders WHERE rowid BETWEEN value1 and value2;
```

Parallelizing the extraction of complex SQL queries is sometimes possible, although the process of breaking a single complex query into multiple components can be challenging. In particular, the coordination of independent processes to guarantee a globally consistent view can be difficult. Unlike the SQL*Plus approach, using the external table data pump unload functionality provides transparent parallel capabilities.

Note that all parallel techniques can use considerably more CPU and I/O resources on the source system, and the impact on the source system should be evaluated before parallelizing any extraction technique.

Extracting into Flat Files Using OCI or Pro*C Programs

OCI programs (or other programs using Oracle call interfaces, such as Pro*C programs), can also be used to extract data. These techniques typically provide improved performance over the SQL*Plus approach, although they also require additional programming. Like the SQL*Plus approach, an OCI program can extract the results of any SQL query. Furthermore, the parallelization techniques described for the SQL*Plus approach can be readily applied to OCI programs as well.

When using OCI or SQL*Plus for extraction, you need additional information besides the data itself. At minimum, you need information about the extracted columns. It is also helpful to know the extraction format, which might be the separator between distinct columns.

Exporting into Export Files Using the Export Utility

The Export utility allows tables (including data) to be exported into Oracle Database export files. Unlike the SQL*Plus and OCI approaches, which describe the extraction of the results of a SQL statement, Export provides a mechanism for extracting database objects. Thus, Export differs from the previous approaches in several important ways:

- The export files contain metadata as well as data. An export file contains not only the raw data of a table, but also information on how to re-create the table, potentially including any indexes, constraints, grants, and other attributes associated with that table.
- A single export file may contain a subset of a single object, many database objects, or even an entire schema.
- Export cannot be directly used to export the results of a complex SQL query. Export can be used only to extract subsets of distinct database objects.
- The output of the Export utility must be processed using the Import utility.

Oracle provides the original Export and Import utilities for backward compatibility and the data pump export/import infrastructure for high-performant, scalable and parallel extraction. See *Oracle Database Utilities* for further details.

Extracting into Export Files Using External Tables

In addition to the Export Utility, you can use external tables to extract the results from any `SELECT` operation. The data is stored in the platform independent, Oracle-internal data pump format and can be processed as regular external table on the target system. The following example extracts the result of a join operation in parallel into the four specified files. The only allowed external table type for extracting data is the Oracle-internal format `ORACLE_DATAPUMP`.

```
CREATE DIRECTORY def_dir AS '/net/dlsun48/private/hbaer/WORK/FEATURES/et';
DROP TABLE extract_cust;
CREATE TABLE extract_cust
ORGANIZATION EXTERNAL
(TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY def_dir ACCESS PARAMETERS
(NOBADFILE NOLOGFILE)
LOCATION ('extract_cust1.exp', 'extract_cust2.exp', 'extract_cust3.exp',
          'extract_cust4.exp'))
PARALLEL 4 REJECT LIMIT UNLIMITED AS
SELECT  c.*, co.country_name, co.country_subregion, co.country_region
FROM customers c, countries co where co.country_id=c.country_id;
```

The total number of extraction files specified limits the maximum degree of parallelism for the write operation. Note that the parallelizing of the extraction does not automatically parallelize the SELECT portion of the statement.

Unlike using any kind of export/import, the metadata for the external table is not part of the created files when using the external table data pump unload. To extract the appropriate metadata for the external table, use the DBMS_METADATA package, as illustrated in the following statement:

```
SET LONG 2000
SELECT DBMS_METADATA.GET_DDL('TABLE','EXTRACT_CUST') FROM DUAL;
```

Extraction Through Distributed Operations

Using distributed-query technology, one Oracle database can directly query tables located in various different source systems, such as another Oracle database or a legacy system connected with the Oracle gateway technology. Specifically, a data warehouse or staging database can directly access tables and data located in a connected source system. Gateways are another form of distributed-query technology. Gateways allow an Oracle database (such as a data warehouse) to access database tables stored in remote, non-Oracle databases. This is the simplest method for moving data between two Oracle databases because it combines the extraction and transformation into a single step, and requires minimal programming. However, this is not always feasible.

Suppose that you wanted to extract a list of employee names with department names from a source database and store this data into the data warehouse. Using an Oracle Net connection and distributed-query technology, this can be achieved using a single SQL statement:

```
CREATE TABLE country_city AS SELECT distinct t1.country_name, t2.cust_city
FROM countries@source_db t1, customers@source_db t2
WHERE t1.country_id = t2.country_id
AND t1.country_name='United States of America';
```

This statement creates a local table in a data mart, country_city, and populates it with data from the countries and customers tables on the source system.

This technique is ideal for moving small volumes of data. However, the data is transported from the source system to the data warehouse through a single Oracle Net connection. Thus, the scalability of this technique is limited. For larger data volumes, file-based data extraction and transportation techniques are often more scalable and thus more appropriate.

See Also:

- *Oracle Database Heterogeneous Connectivity Administrator's Guide* for more information regarding distributed queries
- *Oracle Database Concepts* for more information regarding distributed queries

14

Transportation in Data Warehouses

The following topics provide information about transporting data into a data warehouse:

- Overview of Transportation in Data Warehouses
- Introduction to Transportation Mechanisms in Data Warehouses

Overview of Transportation in Data Warehouses

Transportation is the operation of moving data from one system to another system. In a data warehouse environment, the most common requirements for transportation are in moving data from:

- A source system to a staging database or a data warehouse database
- A staging database to a data warehouse
- A data warehouse to a data mart

Transportation is often one of the simpler portions of the ETL process, and can be integrated with other portions of the process. For example, as shown in Chapter 13, "Extraction in Data Warehouses", distributed query technology provides a mechanism for both extracting and transporting data.

Introduction to Transportation Mechanisms in Data Warehouses

You have three basic choices for transporting data in warehouses:

- Transportation Using Flat Files
- Transportation Through Distributed Operations
- Transportation Using Transportable Tablespaces

Transportation Using Flat Files

The most common method for transporting data is by the transfer of flat files, using mechanisms such as FTP or other remote file system access protocols. Data is unloaded or exported from the source system into flat files using techniques discussed in Chapter 13, "Extraction in Data Warehouses", and is then transported to the target platform using FTP or similar mechanisms.

Because source systems and data warehouses often use different operating systems and database systems, using flat files is often the simplest way to exchange data between heterogeneous systems with minimal transformations. However, even when

transporting data between homogeneous systems, flat files are often the most efficient and most easy-to-manage mechanism for data transfer.

Transportation Through Distributed Operations

Distributed queries, either with or without gateways, can be an effective mechanism for extracting data. These mechanisms also transport the data directly to the target systems, thus providing both extraction and transformation in a single step. Depending on the tolerable impact on time and system resources, these mechanisms can be well suited for both extraction and transformation.

As opposed to flat file transportation, the success or failure of the transportation is recognized immediately with the result of the distributed query or transaction. See Chapter 13, "Extraction in Data Warehouses" for further information.

Transportation Using Transportable Tablespaces

Oracle transportable tablespaces are the fastest way for moving large volumes of data between two Oracle databases. Previous to the introduction of transportable tablespaces, the most scalable data transportation mechanisms relied on moving flat files containing raw data. These mechanisms required that data be unloaded or exported into files from the source database, Then, after transportation, these files were loaded or imported into the target database. Transportable tablespaces entirely bypass the unload and reload steps.

Using transportable tablespaces, Oracle data files (containing table data, indexes, and almost every other Oracle database object) can be directly transported from one database to another. Furthermore, like import and export, transportable tablespaces provide a mechanism for transporting metadata in addition to transporting data.

Transportable tablespaces have some limitations: source and target systems must be running Oracle8*i* (or higher), must use compatible character sets, and, before Oracle Database 10*g*, must run on the same operating system. For details on how to transport tablespace between operating systems, see *Oracle Database Administrator's Guide*.

The most common applications of transportable tablespaces in data warehouses are in moving data from a staging database to a data warehouse, or in moving data from a data warehouse to a data mart.

Transportable Tablespaces Example

Suppose that you have a data warehouse containing sales data, and several data marts that are refreshed monthly. Also suppose that you are going to move one month of sales data from the data warehouse to the data mart.

Step 1 Place the Data to be Transported into its own Tablespace

The current month's data must be placed into a separate tablespace in order to be transported. In this example, you have a tablespace `ts_temp_sales`, which holds a copy of the current month's data. Using the `CREATE TABLE ... AS SELECT` statement, the current month's data can be efficiently copied to this tablespace:

```
CREATE TABLE temp_jan_sales NOLOGGING TABLESPACE ts_temp_sales
AS SELECT * FROM sales
WHERE time_id BETWEEN '31-DEC-1999' AND '01-FEB-2000';
```

Following this operation, the tablespace `ts_temp_sales` is set to read-only:

```
ALTER TABLESPACE ts_temp_sales READ ONLY;
```

A tablespace cannot be transported unless there are no active transactions modifying the tablespace. Setting the tablespace to read-only enforces this.

The tablespace `ts_temp_sales` may be a tablespace that has been especially created to temporarily store data for use by the transportable tablespace features. Following "Copy the Datafiles and Export File to the Target System", this tablespace can be set to read/write, and, if desired, the table `temp_jan_sales` can be dropped, or the tablespace can be re-used for other transportations or for other purposes.

In a given transportable tablespace operation, all of the objects in a given tablespace are transported. Although only one table is being transported in this example, the tablespace `ts_temp_sales` could contain multiple tables. For example, perhaps the data mart is refreshed not only with the new month's worth of sales transactions, but also with a new copy of the customer table. Both of these tables could be transported in the same tablespace. Moreover, this tablespace could also contain other database objects such as indexes, which would also be transported.

Additionally, in a given transportable-tablespace operation, multiple tablespaces can be transported at the same time. This makes it easier to move very large volumes of data between databases. Note, however, that the transportable tablespace feature can only transport a set of tablespaces which contain a complete set of database objects without dependencies on other tablespaces. For example, an index cannot be transported without its table, nor can a partition be transported without the rest of the table. You can use the `DBMS_TTS` package to check that a tablespace is transportable.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `DBMS_TTS` package

In this step, we have copied the January sales data into a separate tablespace; however, in some cases, it may be possible to leverage the transportable tablespace feature without even moving data to a separate tablespace. If the sales table has been partitioned by month in the data warehouse and if each partition is in its own tablespace, then it may be possible to directly transport the tablespace containing the January data. Suppose the January partition, `sales_jan2000`, is located in the tablespace `ts_sales_jan2000`. Then the tablespace `ts_sales_jan2000` could potentially be transported, rather than creating a temporary copy of the January sales data in the `ts_temp_sales`.

However, the same conditions must be satisfied in order to transport the tablespace `ts_sales_jan2000` as are required for the specially created tablespace. First, this tablespace must be set to `READ ONLY`. Second, because a single partition of a partitioned table cannot be transported without the remainder of the partitioned table also being transported, it is necessary to exchange the January partition into a separate table (using the `ALTER TABLE` statement) to transport the January data. The `EXCHANGE` operation is very quick, but the January data will no longer be a part of the underlying `sales` table, and thus may be unavailable to users until this data is exchanged back into the `sales` table after the export of the metadata. The January data can be exchanged back into the `sales` table after you complete step 3.

Step 2 Export the Metadata

The Export utility is used to export the metadata describing the objects contained in the transported tablespace. For our example scenario, the Export command could be:

```
EXP TRANSPORT_TABLESPACE=y TABLESPACES=ts_temp_sales FILE=jan_sales.dmp
```

This operation generates an export file, `jan_sales.dmp`. The export file is small, because it contains only metadata. In this case, the export file contains information describing the table `temp_jan_sales`, such as the column names, column data type,

and all other information that the target Oracle database needs in order to access the objects in ts_temp_sales.

Step 3 Copy the Datafiles and Export File to the Target System

Copy the data files that make up ts_temp_sales, as well as the export file jan_sales.dmp to the data mart platform, using any transportation mechanism for flat files. Once the datafiles have been copied, the tablespace ts_temp_sales can be set to READ WRITE mode if desired.

Step 4 Import the Metadata

Once the files have been copied to the data mart, the metadata should be imported into the data mart:

```
IMP TRANSPORT_TABLESPACE=y DATAFILES='/db/tempjan.f'
    TABLESPACES=ts_temp_sales FILE=jan_sales.dmp
```

At this point, the tablespace ts_temp_sales and the table temp_sales_jan are accessible in the data mart. You can incorporate this new data into the data mart's tables.

You can insert the data from the temp_sales_jan table into the data mart's sales table in one of two ways:

```
INSERT /*+ APPEND */ INTO sales SELECT * FROM temp_sales_jan;
```

Following this operation, you can delete the temp_sales_jan table (and even the entire ts_temp_sales tablespace).

Alternatively, if the data mart's sales table is partitioned by month, then the new transported tablespace and the temp_sales_jan table can become a permanent part of the data mart. The temp_sales_jan table can become a partition of the data mart's sales table:

```
ALTER TABLE sales ADD PARTITION sales_00jan VALUES
  LESS THAN (TO_DATE('01-feb-2000','dd-mon-yyyy'));
ALTER TABLE sales EXCHANGE PARTITION sales_00jan
  WITH TABLE temp_sales_jan INCLUDING INDEXES WITH VALIDATION;
```

Other Uses of Transportable Tablespaces

The previous example illustrates a typical scenario for transporting data in a data warehouse. However, transportable tablespaces can be used for many other purposes. In a data warehousing environment, transportable tablespaces should be viewed as a utility (much like Import/Export or SQL*Loader), whose purpose is to move large volumes of data between Oracle databases. When used in conjunction with parallel data movement operations such as the CREATE TABLE ... AS SELECT and INSERT ... AS SELECT statements, transportable tablespaces provide an important mechanism for quickly transporting data for many purposes.

15

Loading and Transformation

This chapter helps you create and manage a data warehouse, and discusses:

Overview of Loading and Transformation in Data Warehouses

Data transformations are often the most complex and, in terms of processing time, the most costly part of the extraction, transformation, and loading (ETL) process. They can range from simple data conversions to extremely complex data scrubbing techniques. Many, if not all, data transformations can occur within an Oracle database, although transformations are often implemented outside of the database (for example, on flat files) as well.

This chapter introduces techniques for implementing scalable and efficient data transformations within the Oracle Database. The examples in this chapter are relatively simple. Real-world data transformations are often considerably more complex. However, the transformation techniques introduced in this chapter meet the majority of real-world data transformation requirements, often with more scalability and less programming than alternative approaches.

This chapter does not seek to illustrate all of the typical transformations that would be encountered in a data warehouse, but to demonstrate the types of fundamental technology that can be applied to implement these transformations and to provide guidance in how to choose the best techniques.

Transformation Flow

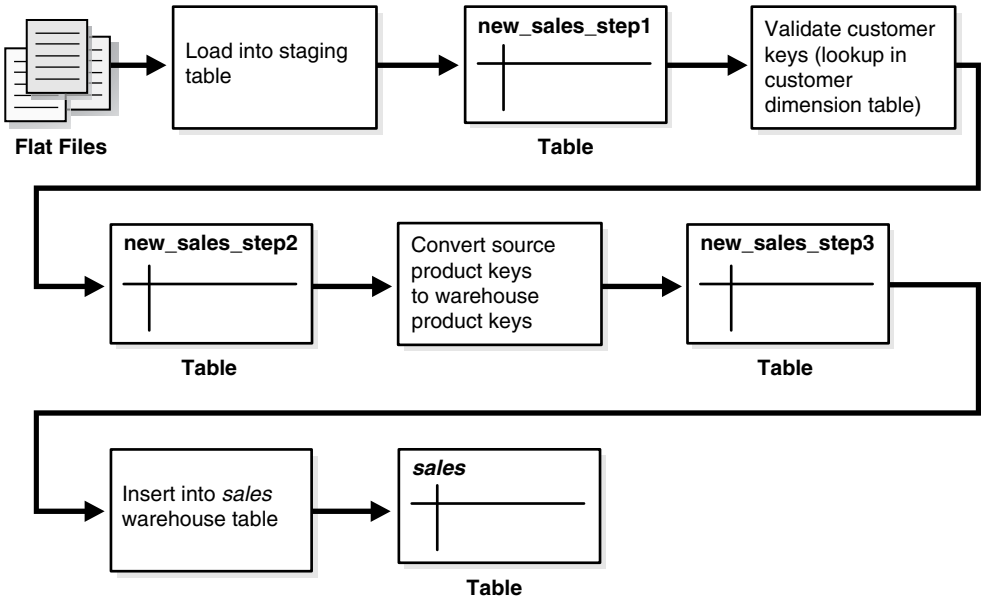
From an architectural perspective, you can transform your data in two ways:

Multistage Data Transformation

The data transformation logic for most data warehouses consists of multiple steps. For example, in transforming new records to be inserted into a sales table, there may be separate logical transformation steps to validate each dimension key.

Figure 15–1 offers a graphical way of looking at the transformation logic.

Figure 15–1 Multistage Data Transformation



When using Oracle Database as a transformation engine, a common strategy is to implement each transformation as a separate SQL operation and to create a separate, temporary staging table (such as the tables `new_sales_step1` and `new_sales_step2` in Figure 15–1) to store the incremental results for each step. This load-then-transform strategy also provides a natural checkpointing scheme to the entire transformation process, which enables the process to be more easily monitored and restarted. However, a disadvantage to multistaging is that the space and time requirements increase.

It may also be possible to combine many simple logical transformations into a single SQL statement or single PL/SQL procedure. Doing so may provide better performance than performing each step independently, but it may also introduce difficulties in modifying, adding, or dropping individual transformations, as well as recovering from failed transformations.

Pipelined Data Transformation

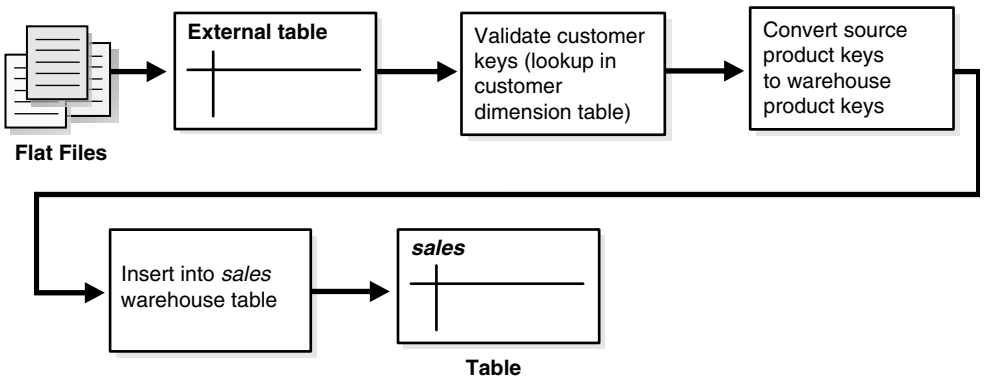
The ETL process flow can be changed dramatically and the database becomes an integral part of the ETL solution.

The new functionality renders some of the former necessary process steps obsolete while some others can be remodeled to enhance the data flow and the data transformation to become more scalable and non-interruptive. The task shifts from serial transform-then-load process (with most of the tasks done outside the database) or load-then-transform process, to an enhanced transform-while-loading.

Oracle offers a wide variety of new capabilities to address all the issues and tasks relevant in an ETL scenario. It is important to understand that the database offers

toolkit functionality rather than trying to address a one-size-fits-all solution. The underlying database has to enable the most appropriate ETL process flow for a specific customer need, and not dictate or constrain it from a technical perspective. Figure 15–2 illustrates the new functionality, which is discussed throughout later sections.

Figure 15–2 Pipelined Data Transformation



Staging Area

The overall speed of your load is determined by how quickly the raw data can be read from the staging area and written to the target table in the database. It is highly recommended that you stage your raw data across as many physical disks as possible to ensure the reading of the raw data is not a bottleneck during the load.

An excellent place to stage the data is in an Oracle Database File System (DBFS). DBFS creates a mountable file system which can be used to access files stored in the database as SecureFile LOBs. DBFS is similar to NFS in that it provides a shared network file system that looks like a local file system. Oracle recommends that you create the DBFS in a separate database from the data warehouse, and that the file system be mounted using the `DIRECT_IO` option to avoid contention on the system page cache while moving the raw data files in and out of the file system. More information on setting up DBFS can be found in *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Loading Mechanisms

You can use the following mechanisms for loading a data warehouse:

- Loading a Data Warehouse with SQL*Loader
- Loading a Data Warehouse with External Tables
- Loading a Data Warehouse with OCI and Direct-Path APIs
- Loading a Data Warehouse with Export/Import

Loading a Data Warehouse with SQL*Loader

Before any data transformations can occur within the database, the raw data must become accessible for the database. One approach is to load it into the database. Chapter 14, "Transportation in Data Warehouses", discusses several techniques for transporting data to an Oracle data warehouse. Perhaps the most common technique for transporting data is by way of flat files.

SQL*Loader is used to move data from flat files into an Oracle data warehouse. During this data load, SQL*Loader can also be used to implement basic data

transformations. When using direct-path SQL*Loader, basic data manipulation, such as datatype conversion and simple NULL handling, can be automatically resolved during the data load. Most data warehouses use direct-path loading for performance reasons.

The conventional-path loader provides broader capabilities for data transformation than a direct-path loader: SQL functions can be applied to any column as those values are being loaded. This provides a rich capability for transformations during the data load. However, the conventional-path loader is slower than direct-path loader. For these reasons, the conventional-path loader should be considered primarily for loading and transforming smaller amounts of data.

The following is a simple example of a SQL*Loader controlfile to load data into the sales table of the sh sample schema from an external file sh_sales.dat. The external flat file sh_sales.dat consists of sales transaction data, aggregated on a daily level. Not all columns of this external file are loaded into sales. This external file is also used asa source for loading the second fact table of the sh sample schema, which is done using an external table:

The following shows the control file (sh_sales.ctl) loading the sales table:

```
LOAD DATA INFILE sh_sales.dat APPEND INTO TABLE sales
FIELDS TERMINATED BY "|"
(PROD_ID, CUST_ID, TIME_ID, CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, AMOUNT_SOLD)
```

It can be loaded with the following command:

```
$  sqlldr control=sh_sales.ctl direct=true
Username:
Password:
```

Loading a Data Warehouse with External Tables

Another approach for handling external data sources is using external tables. Oracle's external table feature enables you to use external data as a virtual table that can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database. You can then use SQL, PL/SQL, and Java to access the external data.

External tables enable the pipelining of the loading phase with the transformation phase. The transformation process can be merged with the loading process without any interruption of the data streaming. It is no longer necessary to stage the data inside the database for further processing inside the database, such as comparison or transformation. For example, the conversion functionality of a conventional load can be used for a direct-path INSERT AS SELECT statement in conjunction with the SELECT from an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (UPDATE/INSERT/DELETE) are possible and no indexes can be created on them.

External tables are mostly compliant with the existing SQL*Loader functionality and provide superior functionality in most cases. External tables are especially useful for environments where the complete external source has to be joined with existing database objects or when the data has to be transformed in a complex manner. For example, unlike SQL*Loader, you can apply any arbitrary SQL transformation and use the direct path insert method. In addition, you can specify a program to be executed (such as zcat) that processes files (such as compressed data files) and enables Oracle Database to use the output (such as uncompressed data files), which means you can load large amounts of compressed data without first uncompressing it on a disk.

You can create an external table named sales_transactions_ext, representing the structure of the complete sales transaction data, represented in the external file sh_sales.gz. The product department is especially interested in a cost analysis on product and time. We thus create a fact table named cost in the sh schema. The operational source data is the same as for the sales fact table. However, because we are not investigating every dimensional information that is provided, the data in the cost fact table has a coarser granularity than in the sales fact table, for example, all different distribution channels are aggregated.

We cannot load the data into the cost fact table without applying the previously mentioned aggregation of the detailed information, due to the suppression of some of the dimensions.

The external table framework offers a solution to solve this. Unlike SQL*Loader, where you would have to load the data before applying the aggregation, you can combine the loading and transformation within a single SQL DML statement, as shown in the following. You do not have to stage the data temporarily before inserting into the target table.

The object directories must already exist, and point to the directory containing the sh_sales.gz file as well as the directory containing the bad and log files.

```
CREATE TABLE sales_transactions_ext
(PROD_ID NUMBER, CUST_ID NUMBER,
 TIME_ID DATE, CHANNEL_ID NUMBER,
 PROMO_ID NUMBER, QUANTITY_SOLD NUMBER,
 AMOUNT_SOLD NUMBER(10,2), UNIT_COST NUMBER(10,2),
 UNIT_PRICE NUMBER(10,2))
ORGANIZATION external (TYPE oracle_loader
  DEFAULT DIRECTORY data_file_dir ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    PREPROCESSOR EXECDIR:'zcat'
    BADFILE log_file_dir:'sh_sales.bad_xt'
    LOGFILE log_file_dir:'sh_sales.log_xt'
    FIELDS TERMINATED BY "|" LDRTRIM
    ( PROD_ID, CUST_ID,
      TIME_ID         DATE(10) "YYYY-MM-DD",
      CHANNEL_ID, PROMO_ID, QUANTITY_SOLD, AMOUNT_SOLD,
      UNIT_COST, UNIT_PRICE))
  location ('sh_sales.gz')
)REJECT LIMIT UNLIMITED;
```

The external table can now be used from within the database, accessing some columns of the external data only, grouping the data, and inserting it into the costs fact table:

```
INSERT /*+ APPEND */ INTO COSTS
(TIME_ID, PROD_ID, UNIT_COST, UNIT_PRICE)
SELECT TIME_ID, PROD_ID, AVG(UNIT_COST), AVG(amount_sold/quantity_sold)
FROM sales_transactions_ext GROUP BY time_id, prod_id;
```

See Also:

- *Oracle Database SQL Language Reference* for a complete description of external table syntax
- *Oracle Database Utilities* for usage examples

Loading a Data Warehouse with OCI and Direct-Path APIs

OCI and direct-path APIs are frequently used when the transformation and computation are done outside the database and there is no need for flat file staging.

Loading a Data Warehouse with Export/Import

Export and import are used when the data is inserted as is into the target system. No complex extractions are possible. See Chapter 13, "Extraction in Data Warehouses" for further information.

Transformation Mechanisms

You have the following choices for transforming data inside the database:

- Transforming Data Using SQL
- Transforming Data Using PL/SQL
- Transforming Data Using Table Functions

Transforming Data Using SQL

Once data is loaded into the database, data transformations can be executed using SQL operations. There are four basic techniques for implementing SQL data transformations:

- CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT
- Transforming Data Using UPDATE
- Transforming Data Using MERGE
- Transforming Data Using Multitable INSERT

CREATE TABLE ... AS SELECT And INSERT /*+APPEND*/ AS SELECT

The `CREATE TABLE ... AS SELECT` statement (CTAS) is a powerful tool for manipulating large sets of data. As shown in the following example, many data transformations can be expressed in standard SQL, and CTAS provides a mechanism for efficiently executing a SQL query and storing the results of that query in a new database table. The `INSERT /*+APPEND*/ ... AS SELECT` statement offers the same capabilities with existing database tables.

In a data warehouse environment, CTAS is typically run in parallel using `NOLOGGING` mode for best performance.

A simple and common type of data transformation is data substitution. In a data substitution transformation, some or all of the values of a single column are modified. For example, our `sales` table has a `channel_id` column. This column indicates whether a given sales transaction was made by a company's own sales force (a direct sale) or by a distributor (an indirect sale).

You may receive data from multiple source systems for your data warehouse. Suppose that one of those source systems processes only direct sales, and thus the source system does not know indirect sales channels. When the data warehouse initially receives sales data from this system, all sales records have a `NULL` value for the `sales.channel_id` field. These `NULL` values must be set to the proper key value. For example, you can do this efficiently using a SQL function as part of the insertion into the target sales table statement. The structure of source table `sales_activity_direct` is as follows:

```
DESC sales_activity_direct
Name           Null?    Type
------------   -----    ----------------
SALES_DATE              DATE
PRODUCT_ID              NUMBER
CUSTOMER_ID             NUMBER
PROMOTION_ID            NUMBER
AMOUNT                  NUMBER
QUANTITY                NUMBER
```

The following SQL statement inserts data from `sales_activity_direct` into the `sales` table of the sample schema, using a SQL function to truncate the sales date values to the midnight time and assigning a fixed channel ID of 3.

```
INSERT /*+ APPEND NOLOGGING PARALLEL */
INTO sales SELECT product_id, customer_id, TRUNC(sales_date), 3,
       promotion_id, quantity, amount
FROM sales_activity_direct;
```

Transforming Data Using UPDATE

Another technique for implementing a data substitution is to use an `UPDATE` statement to modify the `sales.channel_id` column. An `UPDATE` provides the correct result. However, if the data substitution transformations require that a very large percentage of the rows (or all of the rows) be modified, then, it may be more efficient to use a CTAS statement than an `UPDATE`.

Transforming Data Using MERGE

Oracle Database's merge functionality extends SQL, by introducing the SQL keyword `MERGE`, in order to provide the ability to update or insert a row conditionally into a table or out of line single table views. Conditions are specified in the `ON` clause. This is, besides pure bulk loading, one of the most common operations in data warehouse synchronization.

Merge Examples The following discusses various implementations of a merge. The examples assume that new data for the dimension table products is propagated to the data warehouse and has to be either inserted or updated. The table `products_delta` has the same structure as `products`.

Example 15–1 Merge Operation Using SQL

```
MERGE INTO products t USING products_delta s
ON (t.prod_id=s.prod_id)
WHEN MATCHED THEN UPDATE SET
  t.prod_list_price=s.prod_list_price, t.prod_min_price=s.prod_min_price
WHEN NOT MATCHED THEN INSERT (prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_status,
  prod_list_price, prod_min_price)
VALUES (s.prod_id, s.prod_name, s.prod_desc, s.prod_subcategory,
  s.prod_subcategory_desc, s.prod_category, s.prod_category_desc,
  s.prod_status, s.prod_list_price, s.prod_min_price);
```

Transforming Data Using Multitable INSERT

Many times, external data sources have to be segregated based on logical attributes for insertion into different target objects. It is also frequent in data warehouse environments to fan out the same source data into several target objects. Multitable inserts provide a new SQL statement for these kinds of transformations, where data

can either end up in several or exactly one target, depending on the business transformation rules. This insertion can be done conditionally based on business rules or unconditionally.

It offers the benefits of the INSERT ... SELECT statement when multiple tables are involved as targets. In doing so, it avoids the drawbacks of the two obvious alternatives. You either had to deal with *n* independent INSERT … SELECT statements, thus processing the same source data *n* times and increasing the transformation workload *n* times. Alternatively, you had to choose a procedural approach with a per-row determination how to handle the insertion. This solution lacked direct access to high-speed access paths available in SQL.

As with the existing INSERT ... SELECT statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Example 15–2 Unconditional Insert

The following statement aggregates the transactional sales information, stored in sales_activity_direct, on a daily basis and inserts into both the sales and the costs fact table for the current day.

```
INSERT ALL
   INTO sales VALUES (product_id, customer_id, today, 3, promotion_id,
                      quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, promotion_id, 3,
                      product_cost, product_price)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
  s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
  quantity_per_day, p.prod_min_price*0.8 AS product_cost, p.prod_list_price
  AS product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id, s.promotion_id,
  p.prod_min_price*0.8, p.prod_list_price;
```

Example 15–3 Conditional ALL Insert

The following statement inserts a row into the sales and costs tables for all sales transactions with a valid promotion and stores the information about multiple identical orders of a customer in a separate table cum_sales_activity. It is possible two rows will be inserted for some sales transactions, and none for others.

```
INSERT ALL
WHEN promotion_id IN (SELECT promo_id FROM promotions) THEN
   INTO sales VALUES (product_id, customer_id, today, 3, promotion_id,
                       quantity_per_day, amount_per_day)
   INTO costs VALUES (product_id, today, promotion_id, 3,
                      product_cost, product_price)
WHEN num_of_orders > 1 THEN
   INTO cum_sales_activity VALUES (today, product_id, customer_id,
     promotion_id, quantity_per_day, amount_per_day, num_of_orders)
SELECT TRUNC(s.sales_date) AS today, s.product_id, s.customer_id,
   s.promotion_id, SUM(s.amount) AS amount_per_day, SUM(s.quantity)
   quantity_per_day, COUNT(*) num_of_orders, p.prod_min_price*0.8
   AS product_cost, p.prod_list_price AS product_price
FROM sales_activity_direct s, products p
WHERE s.product_id = p.prod_id
AND TRUNC(sales_date) = TRUNC(SYSDATE)
GROUP BY TRUNC(sales_date), s.product_id, s.customer_id,
 s.promotion_id, p.prod_min_price*0.8, p.prod_list_price;
```

Example 15–4 Conditional FIRST Insert

The following statement inserts into an appropriate shipping manifest according to the total quantity and the weight of a product order. An exception is made for high value orders, which are also sent by express, unless their weight classification is not too high. All incorrect orders, in this simple example represented as orders without a quantity, are stored in a separate table. It assumes the existence of appropriate tables `large_freight_shipping`, `express_shipping`, `default_shipping`, and `incorrect_sales_order`.

```
INSERT FIRST WHEN (sum_quantity_sold > 10 AND prod_weight_class < 5) AND
sum_quantity_sold >=1) OR (sum_quantity_sold > 5 AND prod_weight_class > 5) THEN
  INTO large_freight_shipping VALUES
      (time_id, cust_id, prod_id, prod_weight_class, sum_quantity_sold)
  WHEN sum_amount_sold > 1000 AND sum_quantity_sold >=1 THEN
  INTO express_shipping VALUES
      (time_id, cust_id, prod_id, prod_weight_class,
       sum_amount_sold, sum_quantity_sold)
WHEN (sum_quantity_sold >=1) THEN INTO default_shipping VALUES
      (time_id, cust_id, prod_id, sum_quantity_sold)
ELSE INTO incorrect_sales_order VALUES (time_id, cust_id, prod_id)
SELECT s.time_id, s.cust_id, s.prod_id, p.prod_weight_class,
       SUM(amount_sold) AS sum_amount_sold,
       SUM(quantity_sold) AS sum_quantity_sold
FROM sales s, products p
WHERE s.prod_id = p.prod_id AND s.time_id = TRUNC(SYSDATE)
GROUP BY s.time_id, s.cust_id, s.prod_id, p.prod_weight_class;
```

Example 15–5 Mixed Conditional and Unconditional Insert

The following example inserts new customers into the `customers` table and stores all new customers with `cust_credit_limit` higher then 4500 in an additional, separate table for further promotions.

```
INSERT FIRST WHEN cust_credit_limit >= 4500 THEN INTO customers
  INTO customers_special VALUES (cust_id, cust_credit_limit)
  ELSE INTO customers
SELECT * FROM customers_new;
```

See Also:

- Chapter 16, "Maintaining the Data Warehouse" for more information regarding `MERGE` operations

Transforming Data Using PL/SQL

In a data warehouse environment, you can use procedural languages such as PL/SQL to implement complex transformations in the Oracle Database. Whereas CTAS operates on entire tables and emphasizes parallelism, PL/SQL provides a row-based approached and can accommodate very sophisticated transformation rules. For example, a PL/SQL procedure could open multiple cursors and read data from multiple source tables, combine this data using complex business rules, and finally insert the transformed data into one or more target table. It would be difficult or impossible to express the same sequence of operations using standard SQL statements.

Using a procedural language, a specific transformation (or number of transformation steps) within a complex ETL processing can be encapsulated, reading data from an intermediate staging area and generating a new table object as output. A previously generated transformation input table and a subsequent transformation will consume the table generated by this specific transformation. Alternatively, these encapsulated

transformation steps within the complete ETL process can be integrated seamlessly, thus streaming sets of rows between each other without the necessity of intermediate staging. You can use table functions to implement such behavior.

Transforming Data Using Table Functions

Table functions provide the support for pipelined and parallel execution of transformations implemented in PL/SQL, C, or Java. Scenarios as mentioned earlier can be done without requiring the use of intermediate staging tables, which interrupt the data flow through various transformations steps.

What is a Table Function?

A table function is defined as a function that can produce a set of rows as output. Additionally, table functions can take a set of rows as input. Prior to Oracle9*i*, PL/SQL functions:

- Could not take cursors as input.
- Could not be parallelized or pipelined.

Now, functions are not limited in these ways. Table functions extend database functionality by allowing:

- Multiple rows to be returned from a function.
- Results of SQL subqueries (that select multiple rows) to be passed directly to functions.
- Functions take cursors as input.
- Functions can be parallelized.
- Returning result sets incrementally for further processing as soon as they are created. This is called incremental pipelining

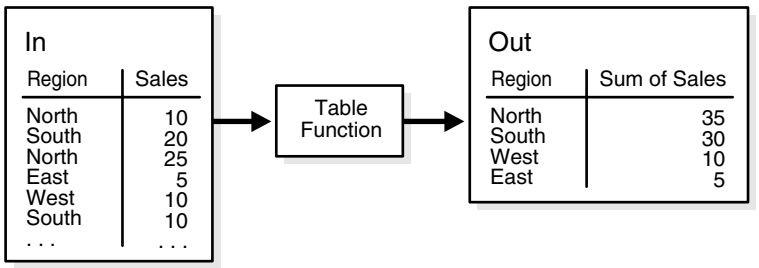
Table functions can be defined in PL/SQL using a native PL/SQL interface, or in Java or C using the Oracle Data Cartridge Interface (ODCI).

See Also:

- *Oracle Database PL/SQL Language Reference*
- *Oracle Database Data Cartridge Developer's Guide*

Figure 15–3 illustrates a typical aggregation where you input a set of rows and output a set of rows, in that case, after performing a `SUM` operation.

Figure 15–3 Table Function Example



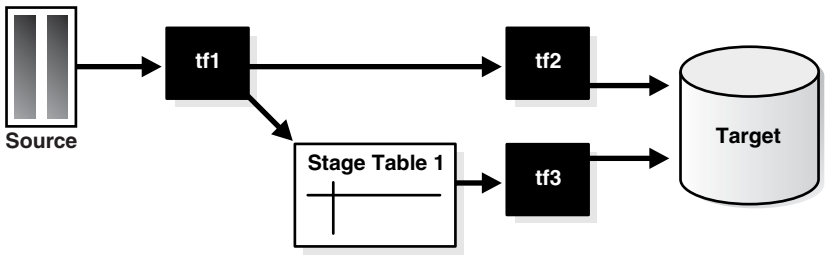
The pseudocode for this operation would be similar to:

```
INSERT INTO Out SELECT * FROM ("Table Function"(SELECT * FROM In));
```

The table function takes the result of the `SELECT` on `In` as input and delivers a set of records in a different format as output for a direct insertion into `Out`.

Additionally, a table function can fan out data within the scope of an atomic transaction. This can be used for many occasions like an efficient logging mechanism or a fan out for other independent transformations. In such a scenario, a single staging table is needed.

Figure 15–4 Pipelined Parallel Transformation with Fanout



The pseudocode for this would be similar to:

```
INSERT INTO target SELECT * FROM (tf2(SELECT *
FROM (tf1(SELECT * FROM source))));
```

This inserts into `target` and, as part of `tf1`, into `Stage Table 1` within the scope of an atomic transaction.

```
INSERT INTO target SELECT * FROM tf3(SELT * FROM stage_table1);
```

Example 15–6 Table Functions Fundamentals

The following examples demonstrate the fundamentals of table functions, without the usage of complex business rules implemented inside those functions. They are chosen for demonstration purposes only, and are all implemented in PL/SQL.

Table functions return sets of records and can take cursors as input. Besides the `sh` sample schema, you have to set up the following database objects before using the examples:

```
CREATE TYPE product_t AS OBJECT (
      prod_id                 NUMBER(6)
    , prod_name               VARCHAR2(50)
    , prod_desc               VARCHAR2(4000)
    , prod_subcategory        VARCHAR2(50)
    , prod_subcategory_desc   VARCHAR2(2000)
    , prod_category           VARCHAR2(50)
    , prod_category_desc      VARCHAR2(2000)
    , prod_weight_class       NUMBER(2)
    , prod_unit_of_measure    VARCHAR2(20)
    , prod_pack_size          VARCHAR2(30)
    , supplier_id             NUMBER(6)
    , prod_status             VARCHAR2(20)
    , prod_list_price         NUMBER(8,2)
    , prod_min_price          NUMBER(8,2)
);
/
CREATE TYPE product_t_table AS TABLE OF product_t;
/
```

```
COMMIT;

CREATE OR REPLACE PACKAGE cursor_PKG AS
  TYPE product_t_rec IS RECORD (
      prod_id                   NUMBER(6)
    , prod_name                 VARCHAR2(50)
    , prod_desc                 VARCHAR2(4000)
    , prod_subcategory          VARCHAR2(50)
    , prod_subcategory_desc     VARCHAR2(2000)
    , prod_category             VARCHAR2(50)
    , prod_category_desc        VARCHAR2(2000)
    , prod_weight_class         NUMBER(2)
    , prod_unit_of_measure      VARCHAR2(20)
    , prod_pack_size            VARCHAR2(30)
    , supplier_id               NUMBER(6)
    , prod_status               VARCHAR2(20)
    , prod_list_price           NUMBER(8,2)
    , prod_min_price            NUMBER(8,2));
  TYPE product_t_rectab IS TABLE OF product_t_rec;
  TYPE strong_refcur_t IS REF CURSOR RETURN product_t_rec;
  TYPE refcur_t IS REF CURSOR;
END;
/

REM artificial help table, used later
CREATE TABLE obsolete_products_errors (prod_id NUMBER, msg VARCHAR2(2000));
```

The following example demonstrates a simple filtering; it shows all obsolete products except the `prod_category` Electronics. The table function returns the result set as a set of records and uses a weakly typed `REF CURSOR` as input.

```
CREATE OR REPLACE FUNCTION obsolete_products(cur cursor_pkg.refcur_t)
RETURN product_t_table
IS
    prod_id                   NUMBER(6);
    prod_name                 VARCHAR2(50);
    prod_desc                 VARCHAR2(4000);
    prod_subcategory          VARCHAR2(50);
    prod_subcategory_desc     VARCHAR2(2000);
    prod_category             VARCHAR2(50);
    prod_category_desc        VARCHAR2(2000);
    prod_weight_class         NUMBER(2);
    prod_unit_of_measure      VARCHAR2(20);
    prod_pack_size            VARCHAR2(30);
    supplier_id               NUMBER(6);
    prod_status               VARCHAR2(20);
    prod_list_price           NUMBER(8,2);
    prod_min_price            NUMBER(8,2);
    sales NUMBER:=0;
    objset product_t_table := product_t_table();
    i NUMBER := 0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
    prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
    prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
    prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     -- Category Electronics is not meant to be obsolete and will be suppressed
```

```
     IF prod_status='obsolete' AND prod_category != 'Electronics' THEN
     -- append to collection
     i:=i+1;
     objset.extend;
     objset(i):=product_t( prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcategory_desc, prod_category, prod_category_desc,
     prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
     prod_status, prod_list_price, prod_min_price);
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN objset;
END;
/
```

You can use the table function in a SQL statement to show the results. Here we use additional SQL functionality for the output:

```
SELECT DISTINCT UPPER(prod_category), prod_status
FROM TABLE(obsolete_products(
   CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
   prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
   prod_unit_of_measure, prod_pack_size,
   supplier_id, prod_status, prod_list_price, prod_min_price
          FROM products)));
```

The following example implements the same filtering than the first one. The main differences between those two are:

- This example uses a strong typed `REF CURSOR` as input and can be parallelized based on the objects of the strong typed cursor, as shown in one of the following examples.
- The table function returns the result set incrementally as soon as records are created.

```
CREATE OR REPLACE FUNCTION
  obsolete_products_pipe(cur cursor_pkg.strong_refcur_t) RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    prod_id                 NUMBER(6);
    prod_name               VARCHAR2(50);
    prod_desc               VARCHAR2(4000);
    prod_subcategory        VARCHAR2(50);
    prod_subcategory_desc   VARCHAR2(2000);
    prod_category           VARCHAR2(50);
    prod_category_desc      VARCHAR2(2000);
    prod_weight_class       NUMBER(2);
    prod_unit_of_measure    VARCHAR2(20);
    prod_pack_size          VARCHAR2(30);
    supplier_id             NUMBER(6);
    prod_status             VARCHAR2(20);
    prod_list_price         NUMBER(8,2);
    prod_min_price          NUMBER(8,2);
    sales NUMBER:=0;
BEGIN
 LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
     prod_subcategory_desc, prod_category, prod_category_desc,
     prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
```

```
    prod_status, prod_list_price, prod_min_price;
    EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
    IF prod_status='obsolete' AND prod_category !='Electronics' THEN
      PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory,
 prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
 prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
 prod_list_price, prod_min_price));
    END IF;
  END LOOP;
  CLOSE cur;
  RETURN;
END;
/
```

You can use the table function as follows:

```
SELECT DISTINCT prod_category,
                DECODE(prod_status,'obsolete','NO LONGER AVAILABLE','N/A')
FROM TABLE(obsolete_products_pipe(
  CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
         prod_subcategory_desc, prod_category, prod_category_desc,
         prod_weight_class, prod_unit_of_measure, prod_pack_size,
         supplier_id, prod_status, prod_list_price, prod_min_price
         FROM products)));
```

We now change the degree of parallelism for the input table products and issue the same statement again:

```
ALTER TABLE products PARALLEL 4;
```

The session statistics show that the statement has been parallelized:

```
SELECT * FROM V$PQ_SESSTAT WHERE statistic='Queries Parallelized';

STATISTIC              LAST_QUERY  SESSION_TOTAL
--------------------   ----------  -------------
Queries Parallelized            1              3

1 row selected.
```

Table functions are also capable to fanout results into persistent table structures. This is demonstrated in the next example. The function filters returns all obsolete products except a those of a specific `prod_category` (default Electronics), which was set to status `obsolete` by error. The result set of the table function consists of all other obsolete product categories. The detected wrong `prod_id` IDs are stored in a separate table structure `obsolete_products_error`. Note that if a table function is part of an autonomous transaction, it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement to avoid an error in the callings subprogram. Its result set consists of all other obsolete product categories. It furthermore demonstrates how normal variables can be used in conjunction with table functions:

```
CREATE OR REPLACE FUNCTION obsolete_products_dml(cur cursor_pkg.strong_refcur_t,
 prod_cat varchar2 DEFAULT 'Electronics') RETURN product_t_table
PIPELINED
PARALLEL_ENABLE (PARTITION cur BY ANY) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    prod_id                   NUMBER(6);
    prod_name                 VARCHAR2(50);
    prod_desc                 VARCHAR2(4000);
    prod_subcategory          VARCHAR2(50);
    prod_subcategory_desc     VARCHAR2(2000);
```

```
    prod_category             VARCHAR2(50);
    prod_category_desc        VARCHAR2(2000);
    prod_weight_class         NUMBER(2);
    prod_unit_of_measure      VARCHAR2(20);
    prod_pack_size            VARCHAR2(30);
    supplier_id               NUMBER(6);
    prod_status               VARCHAR2(20);
    prod_list_price           NUMBER(8,2);
    prod_min_price            NUMBER(8,2);
    sales                     NUMBER:=0;
BEGIN
   LOOP
     -- Fetch from cursor variable
     FETCH cur INTO prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
     prod_list_price, prod_min_price;
     EXIT WHEN cur%NOTFOUND; -- exit when last row is fetched
     IF prod_status='obsolete' THEN
       IF prod_category=prod_cat THEN
          INSERT INTO obsolete_products_errors VALUES
          (prod_id, 'correction: category '||UPPER(prod_cat)||' still
   available');
          COMMIT;
       ELSE
       PIPE ROW (product_t( prod_id, prod_name, prod_desc, prod_subcategory,
 prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
 prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
 prod_list_price, prod_min_price));
       END IF;
     END IF;
   END LOOP;
   CLOSE cur;
   RETURN;
END;
/
```

The following query shows all obsolete product groups except the `prod_category` Electronics, which was wrongly set to status `obsolete`:

```
SELECT DISTINCT prod_category, prod_status FROM TABLE(obsolete_products_dml(
CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
  prod_list_price, prod_min_price
FROM products)));
```

As you can see, there are some products of the `prod_category` Electronics that were obsoleted by accident:

```
SELECT DISTINCT msg FROM obsolete_products_errors;
```

Taking advantage of the second input variable, you can specify a different product group than Electronics to be considered:

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(
CURSOR(SELECT prod_id, prod_name, prod_desc, prod_subcategory,
  prod_subcategory_desc, prod_category, prod_category_desc, prod_weight_class,
  prod_unit_of_measure, prod_pack_size, supplier_id, prod_status,
  prod_list_price, prod_min_price
```

```
FROM products),'Photo'));
```

Because table functions can be used like a normal table, they can be nested, as shown in the following:

```
SELECT DISTINCT prod_category, prod_status
FROM TABLE(obsolete_products_dml(CURSOR(SELECT *
FROM TABLE(obsolete_products_pipe(CURSOR(SELECT prod_id, prod_name, prod_desc,
 prod_subcategory, prod_subcategory_desc, prod_category, prod_category_desc,
 prod_weight_class, prod_unit_of_measure, prod_pack_size, supplier_id,
 prod_status, prod_list_price, prod_min_price
FROM products))))));
```

The biggest advantage of Oracle Database's ETL is its toolkit functionality, where you can combine any of the latter discussed functionality to improve and speed up your ETL processing. For example, you can take an external table as input, join it with an existing table and use it as input for a parallelized table function to process complex business logic. This table function can be used as input source for a `MERGE` operation, thus streaming the new information for the data warehouse, provided in a flat file within one single statement through the complete ETL process.

See *Oracle Database PL/SQL Language Reference* for details about table functions and the PL/SQL programming. For details about table functions implemented in other languages, see *Oracle Database Data Cartridge Developer's Guide*.

Error Logging and Handling Mechanisms

Having data that is not clean is very common when loading and transforming data, especially when dealing with data coming from a variety of sources, including external ones. If this dirty data causes you to abort a long-running load or transformation operation, a lot of time and resources is wasted. The following sections discuss the two main causes of errors and how to address them:

- Business Rule Violations
- Data Rule Violations (Data Errors)

Business Rule Violations

Data that is logically not clean violates business rules that are known prior to any data consumption. Most of the time, handling these kind of errors will be incorporated into the loading or transformation process. However, in situations where the error identification for all records would become too expensive and the business rule can be enforced as a data rule violation, for example, testing hundreds of columns to see if they are `NOT NULL`, programmers often choose to handle even known possible logical error cases more generically. An example of this is shown in "Data Error Scenarios" on page 15-20.

Incorporating logical rules can be as easy as applying filter conditions on the data input stream or as complex as feeding the dirty data into a different transformation workflow. Some examples are as follows:

- Filtering of logical data errors using SQL. Data that does not adhere to certain conditions is filtered out prior to being processed.
- Identifying and separating logical data errors. In simple cases, this can be accomplished using SQL, as shown in Example 15–1, "Merge Operation Using SQL", or in more complex cases in a procedural approach, as shown in Example 15–6, "Table Functions Fundamentals".

Data Rule Violations (Data Errors)

Unlike logical errors, data rule violations are not usually anticipated by the load or transformation process. Such unexpected data rule violations (also known as data errors) that are not handled from an operation cause the operation to fail. Data rule violations are error conditions that happen inside the database and cause a statement to fail. Examples of this are data type conversion errors or constraint violations.

In the past, SQL did not offer a way to handle data errors on a row level as part of its bulk processing. The only way to handle data errors inside the database was to use PL/SQL. Now, however, you can log data errors into a special error table while the DML operation continues.

The following sections briefly discuss exception handling with PL/SQL and DML error logging tables.

Handling Data Errors in PL/SQL

The following statement is an example of how error handling can be done using PL/SQL. Note that you have to use procedural record-level processing to catch any errors. This statement is a rough equivalent of the statement discussed in "Handling Data Errors with an Error Logging Table" on page 15-17.

```
DECLARE
errm number default 0;
BEGIN
FOR crec IN (SELECT product_id, customer_id, TRUNC(sales_date) sd,
                  promotion_id, quantity, amount
             FROM sales_activity_direct) loop

BEGIN
 INSERT INTO sales VALUES (crec.product_id, crec.customer_id,
                           crec.sd, 3, crec.promotion_id,
                           crec.quantity, crec.amount);
exception
WHEN others then
 errm := sqlerrm;
 INSERT INTO sales_activity_error
        VALUES (errm, crec.product_id, crec.customer_id, crec.sd,
                crec.promotion_id, crec.quantity, crec.amount);
END;
END loop;
END;
/
```

Handling Data Errors with an Error Logging Table

DML error logging extends existing DML functionality by enabling you to specify the name of an error logging table into which Oracle Database should record errors encountered during DML operations. This enables you to complete the DML operation in spite of any errors, and to take corrective action on the erroneous rows at a later time.

This DML error logging table consists of several mandatory control columns and a set of user-defined columns that represent either all or a subset of the columns of the target table of the DML operation using a data type that is capable of storing potential errors for the target column. For example, you need a `VARCHAR2` data type in the error logging table to store `TO_NUM` data type conversion errors for a `NUMBER` column in the target table. You should use the `DBMS_ERRLOG` package to create the DML error

logging tables. See the *Oracle Database PL/SQL Packages and Types Reference* for more information about this package and the structure of the logging table.

The column name mapping between the DML target table and an error logging table determines which columns besides the control columns is logged for a DML operation.

The following statement illustrates how to enhance the example in "Transforming Data Using SQL" on page 15-6 with DML error logging:

```
INSERT /*+ APPEND PARALLEL */
INTO sales SELECT product_id, customer_id, TRUNC(sales_date), 3,
   promotion_id, quantity, amount
FROM sales_activity_direct
LOG ERRORS INTO sales_activity_errors('load_20040802')
REJECT LIMIT UNLIMITED
```

All data errors are logged into table `sales_activity_errors`, identified by the optional tag `load_20040802`. The `INSERT` statement succeeds even in the presence of data errors. Note that you have to create the DML error logging table prior to using this statement.

If `REJECT LIMIT X` had been specified, the statement would have failed with the error message of `error X=1`. The error message can be different for different reject limits. In the case of a failing statement, only the DML statement is rolled back, not the insertion into the DML error logging table. The error logging table will contain X+1 rows.

A DML error logging table can be in a different schema than the executing user, but you must fully specify the table name in that case. Optionally, the name of the DML error logging table can be omitted; Oracle then assumes a default name for the table as generated by the `DBMS_ERRLOG` package.

Oracle Database logs the following errors during DML operations:

- Column values that are too large.
- Constraint violations (`NOT NULL`, unique, referential, and check constraints).
- Errors raised during trigger execution.
- Errors resulting from type conversion between a column in a subquery and the corresponding column of the table.
- Partition mapping errors.

The following conditions cause the statement to fail and roll back without invoking the error logging capability:

- Violated deferred constraints.
- Out of space errors.
- Any direct-path `INSERT` operation (`INSERT` or `MERGE`) that raises a unique constraint or index violation.
- Any `UPDATE` operation (`UPDATE` or `MERGE`) that raises a unique constraint or index violation.

In addition, you cannot track errors in the error logging table for `LONG`, `LOB`, or object type columns. See *Oracle Database SQL Language Reference* for more information on restrictions when using error logging.

DML error logging can be applied to any kind of DML operation. Several examples are discussed in the following section.

Note that SQL*Loader as an external load utility offers the functionality of logging data errors as well, but lacks the advantage of the integrated ETL processing inside the database.

Loading and Transformation Scenarios

The following sections offer examples of typical loading and transformation tasks:

Key Lookup Scenario

A typical transformation is the key lookup. For example, suppose that sales transaction data has been loaded into a retail data warehouse. Although the data warehouse's `sales` table contains a `product_id` column, the sales transaction data extracted from the source system contains Uniform Price Codes (UPC) instead of product IDs. Therefore, it is necessary to transform the UPC codes into product IDs before the new sales transaction data can be inserted into the `sales` table.

In order to execute this transformation, a lookup table must relate the `product_id` values to the UPC codes. This table might be the `product` dimension table, or perhaps another table in the data warehouse that has been created specifically to support this transformation. For this example, we assume that there is a table named `product`, which has a `product_id` and an `upc_code` column.

This data substitution transformation can be implemented using the following CTAS statement:

```
CREATE TABLE temp_sales_step2 NOLOGGING PARALLEL AS SELECT sales_transaction_id,
  product.product_id sales_product_id, sales_customer_id, sales_time_id,
  sales_channel_id, sales_quantity_sold, sales_dollar_amount
FROM  temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code;
```

This CTAS statement converts each valid UPC code to a valid `product_id` value. If the ETL process has guaranteed that each UPC code is valid, then this statement alone may be sufficient to implement the entire transformation.

Business Rule Violation Scenario

In the preceding example, if you must also handle new sales data that does not have valid UPC codes (a logical data error), you can use an additional CTAS statement to identify the invalid rows:

```
CREATE TABLE temp_sales_step1_invalid NOLOGGING PARALLEL AS
SELECT * FROM temp_sales_step1 s
WHERE NOT EXISTS (SELECT 1 FROM product p WHERE p.upc_code=s.upc_code);
```

This invalid data is now stored in a separate table, `temp_sales_step1_invalid`, and can be handled separately by the ETL process.

Another way to handle invalid data is to modify the original CTAS to use an outer join, as in the following statement:

```
CREATE TABLE temp_sales_step2 NOLOGGING PARALLEL AS
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id, sales_quantity_sold,
   sales_dollar_amount
FROM  temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

Using this outer join, the sales transactions that originally contained invalidated UPC codes are assigned a `product_id` of `NULL`. These transactions can be handled later. Alternatively, you could use a multi-table insert, separating the values with a `product_id` of `NULL` into a separate table; this might be a beneficial approach when the expected error count is relatively small compared to the total data volume. You do not have to touch the large target table but only a small one for a subsequent processing.

```
INSERT /*+ APPEND PARALLEL */ FIRST
WHEN sales_product_id IS NOT NULL THEN
   INTO temp_sales_step2
   VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
ELSE
   INTO temp_sales_step1_invalid
   VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount
FROM temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

Note that for this solution, the empty tables `temp_sales_step2` and `temp_sales_step1_invalid` must already exist.

Additional approaches to handling invalid UPC codes exist. Some data warehouses may choose to insert null-valued `product_id` values into their `sales` table, while others may not allow any new data from the entire batch to be inserted into the `sales` table until all invalid UPC codes have been addressed. The correct approach is determined by the business requirements of the data warehouse. Irrespective of the specific requirements, exception handling can be addressed by the same basic SQL techniques as transformations.

Data Error Scenarios

If the quality of the data is unknown, the example discussed in the preceding section could be enhanced to handle unexpected data errors, for example, data type conversion errors, as shown in the following:

```
INSERT /*+ APPEND PARALLEL */ FIRST
WHEN sales_product_id IS NOT NULL THEN
INTO temp_sales_step2
VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
LOG ERRORS INTO sales_step2_errors('load_20040804')
REJECT LIMIT UNLIMITED
ELSE
INTO temp_sales_step1_invalid
VALUES (sales_transaction_id, sales_product_id,
```

```
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
LOG ERRORS INTO sales_step2_errors('load_20040804')
REJECT LIMIT UNLIMITED
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount
FROM temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+);
```

This statement tracks the logical data error of not having a valid product UPC code in table `temp_sales_step1_invalid` and all other possible errors in a DML error logging table called `sales_step2_errors`. Note that an error logging table can be used for several DML operations.

An alternative to this approach would be to enforce the business rule of having a valid UPC code on the database level with a `NOT NULL` constraint. Using an outer join, all orders not having a valid UPC code would be mapped to a `NULL` value and then treated as data errors. This DML error logging capability is used to track these errors in the following statement:

```
INSERT /*+ APPEND PARALLEL */
INTO temp_sales_step2
VALUES (sales_transaction_id, sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount)
SELECT sales_transaction_id, product.product_id sales_product_id,
   sales_customer_id, sales_time_id, sales_channel_id,
   sales_quantity_sold, sales_dollar_amount
FROM temp_sales_step1, product
WHERE temp_sales_step1.upc_code = product.upc_code (+)
LOG ERRORS INTO sales_step2_errors('load_20040804')
REJECT LIMIT UNLIMITED;
```

The error logging table contains all records that would have caused the DML operation to fail. You can use its content to analyze and correct any error. The content in the error logging table is preserved for any DML operation, irrespective of the success of the DML operation itself. Let us assume the following SQL statement failed because the reject limit was reached:

```
SQL> INSERT /*+ APPEND NOLOGGING PARALLEL */ INTO sales_overall
2 SELECT * FROM sales_activity_direct
3 LOG ERRORS INTO err$_sales_overall ('load_test2')
4 REJECT LIMIT 10;
SELECT * FROM sales_activity_direct
*
ERROR at line 2:
ORA-01722: invalid number
```

The name of the error logging table, `err$_sales_overall`, is the default derived by using the `DBMS_ERRLOG` package. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

The error message raised by Oracle occurs where the first after the error limit is reached. The next error (number 11) is the one that raised an error. The error message that is displayed is based on the error that exceeded the limit, so, for example, the ninth error could be different from the eleventh error.

The target table sales_overall will not show any records being entered (assumed that the table was empty before), but the error logging table will contain 11 rows (REJECT LIMIT + 1)

```
SQL> SELECT COUNT(*) FROM sales_overall;
COUNT(*)
----------
0

SQL> SELECT COUNT(*) FROM err$_sales_overall;
COUNT(*)
----------
11
```

A DML error logging table consists of several fixed control columns that are mandatory for every error logging table. Besides the Oracle error number, Oracle enforces storing the error message as well. In many cases, the error message provides additional information to analyze and resolve the root cause for the data error. The following SQL output of a DML error logging table shows this difference. Note that the second output contains the additional information for rows that were rejected due to NOT NULL violations.

```
SQL> SELECT DISTINCT ora_err_number$ FROM err$_sales_overall;

ORA_ERR_NUMBER$
---------------
           1400
           1722
           1830
           1847

SQL> SELECT DISTINCT ora_err_number$, ora_err_mesg$ FROM err$_sales_overall;

ORA_ERR_NUMBER$       ORA_ERR_MESG$
           1400       ORA-01400: cannot insert NULL into
                      ("SH"."SALES_OVERALL"."CUST_ID")
           1400       ORA-01400: cannot insert NULL into
                      ("SH"."SALES_OVERALL"."PROD_ID")
           1722       ORA-01722: invalid number
           1830       ORA-01830: date format picture ends before
                      converting entire input string
           1847       ORA-01847: day of month must be between 1 and last
                      day of month
```

See *Oracle Database Administrator's Guide* for a detailed description of control columns.

Pivoting Scenarios

A data warehouse can receive data from many different sources. Some of these source systems may not be relational databases and may store data in very different formats from the data warehouse. For example, suppose that you receive a set of sales records from a nonrelational database having the form:

```
product_id, customer_id, weekly_start_date, sales_sun, sales_mon, sales_tue,
  sales_wed, sales_thu, sales_fri, sales_sat
```

The input table looks like the following:

```
SELECT * FROM sales_input_table;
```

```
PRODUCT_ID CUSTOMER_ID WEEKLY_ST  SALES_SUN  SALES_MON  SALES_TUE  SALES_WED SALES_THU  SALES_FRI  SALES_SAT
---------- ----------- ---------  ---------- ---------- ---------- ---------- ---------- ---------- ----------
       111         222 01-OCT-00         100        200        300        400        500        600        700
       222         333 08-OCT-00         200        300        400        500        600        700        800
       333         444 15-OCT-00         300        400        500        600        700        800        900
```

In your data warehouse, you would want to store the records in a more typical relational form in a fact table `sales` of the `sh` sample schema:

```
prod_id, cust_id, time_id, amount_sold
```

> **Note:** A number of constraints on the `sales` table have been disabled for purposes of this example, because the example ignores a number of table columns for the sake of brevity.

Thus, you need to build a transformation such that each record in the input stream must be converted into seven records for the data warehouse's `sales` table. This operation is commonly referred to as **pivoting**, and Oracle Database offers several ways to do this.

The result of the previous example will resemble the following:

```
SELECT prod_id, cust_id, time_id, amount_sold FROM sales;

   PROD_ID    CUST_ID   TIME_ID   AMOUNT_SOLD
---------- ----------   --------- -----------
       111        222   01-OCT-00         100
       111        222   02-OCT-00         200
       111        222   03-OCT-00         300
       111        222   04-OCT-00         400
       111        222   05-OCT-00         500
       111        222   06-OCT-00         600
       111        222   07-OCT-00         700
       222        333   08-OCT-00         200
       222        333   09-OCT-00         300
       222        333   10-OCT-00         400
       222        333   11-OCT-00         500
       222        333   12-OCT-00         600
       222        333   13-OCT-00         700
       222        333   14-OCT-00         800
       333        444   15-OCT-00         300
       333        444   16-OCT-00         400
       333        444   17-OCT-00         500
       333        444   18-OCT-00         600
       333        444   19-OCT-00         700
       333        444   20-OCT-00         800
       333        444   21-OCT-00         900
```

Example 15–7 Pivoting

The following example uses the multitable insert syntax to insert into the demo table `sh.sales` some data from an input table with a different structure. The multitable `INSERT` statement looks like the following:

```
INSERT ALL INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date, sales_sun)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+1, sales_mon)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+2, sales_tue)
```

```
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+3, sales_wed)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+4, sales_thu)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+5, sales_fri)
      INTO sales (prod_id, cust_id, time_id, amount_sold)
      VALUES (product_id, customer_id, weekly_start_date+6, sales_sat)
SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
FROM sales_input_table;
```

This statement only scans the source table once and then inserts the appropriate data for each day.

See Also:

- "Pivoting Operations" on page 22-33 for more information regarding pivoting
- *Oracle Database SQL Language Reference* for `pivot_clause` syntax

16

Maintaining the Data Warehouse

This chapter discusses how to load and refresh a data warehouse, and discusses:

- Using Partitioning to Improve Data Warehouse Refresh
- Optimizing DML Operations During Refresh
- Refreshing Materialized Views
- Using Materialized Views with Partitioned Tables

Using Partitioning to Improve Data Warehouse Refresh

ETL (Extraction, Transformation and Loading) is done on a scheduled basis to reflect changes made to the original source system. During this step, you physically insert the new, clean data into the production data warehouse schema, and take all of the other steps necessary (such as building indexes, validating constraints, taking backups) to make this new data available to the end users. Once all of this data has been loaded into the data warehouse, the materialized views have to be updated to reflect the latest data.

The partitioning scheme of the data warehouse is often crucial in determining the efficiency of refresh operations in the data warehouse load process. In fact, the load process is often the primary consideration in choosing the partitioning scheme of data warehouse tables and indexes.

The partitioning scheme of the largest data warehouse tables (for example, the fact table in a star schema) should be based upon the loading paradigm of the data warehouse.

Most data warehouses are loaded with new data on a regular schedule. For example, every night, week, or month, new data is brought into the data warehouse. The data being loaded at the end of the week or month typically corresponds to the transactions for the week or month. In this very common scenario, the data warehouse is being loaded by time. This suggests that the data warehouse tables should be partitioned on a date column. In our data warehouse example, suppose the new data is loaded into the `sales` table every month. Furthermore, the `sales` table has been partitioned by month. These steps show how the load process will proceed to add the data for a new month (January 2001) to the table `sales`.

1. Place the new data into a separate table, `sales_01_2001`. This data can be directly loaded into `sales_01_2001` from outside the data warehouse, or this data can be the result of previous data transformation operations that have already occurred in the data warehouse. `sales_01_2001` has the exact same columns, datatypes, and so forth, as the `sales` table. Gather statistics on the `sales_01_2001` table.

2. Create indexes and add constraints on sales_01_2001. Again, the indexes and constraints on sales_01_2001 should be identical to the indexes and constraints on sales. Indexes can be built in parallel and should use the NOLOGGING and the COMPUTE STATISTICS options. For example:

```
CREATE BITMAP INDEX sales_01_2001_customer_id_bix
  ON sales_01_2001(customer_id)
      TABLESPACE sales_idx NOLOGGING PARALLEL 8 COMPUTE STATISTICS;
```

   Apply all constraints to the sales_01_2001 table that are present on the sales table. This includes referential integrity constraints. A typical constraint would be:

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_customer_id
      REFERENCES customer(customer_id) ENABLE NOVALIDATE;
```

   If the partitioned table sales has a primary or unique key that is enforced with a global index structure, ensure that the constraint on sales_pk_jan01 is validated without the creation of an index structure, as in the following:

```
ALTER TABLE sales_01_2001 ADD CONSTRAINT sales_pk_jan01
PRIMARY KEY (sales_transaction_id) DISABLE VALIDATE;
```

   The creation of the constraint with ENABLE clause would cause the creation of a unique index, which does not match a local index structure of the partitioned table. You must not have any index structure built on the nonpartitioned table to be exchanged for existing global indexes of the partitioned table. The exchange command would fail.

3. Add the sales_01_2001 table to the sales table.

   In order to add this new data to the sales table, we must do two things. First, we must add a new partition to the sales table. We will use the ALTER TABLE ... ADD PARTITION statement. This will add an empty partition to the sales table:

```
ALTER TABLE sales ADD PARTITION sales_01_2001
VALUES LESS THAN (TO_DATE('01-FEB-2001', 'DD-MON-YYYY'));
```

   Then, we can add our newly created table to this partition using the EXCHANGE PARTITION operation. This will exchange the new, empty partition with the newly loaded table.

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_2001 WITH TABLE sales_01_2001
INCLUDING INDEXES WITHOUT VALIDATION UPDATE GLOBAL INDEXES;
```

   The EXCHANGE operation will preserve the indexes and constraints that were already present on the sales_01_2001 table. For unique constraints (such as the unique constraint on sales_transaction_id), you can use the UPDATE GLOBAL INDEXES clause, as shown previously. This will automatically maintain your global index structures as part of the partition maintenance operation and keep them accessible throughout the whole process. If there were only foreign-key constraints, the exchange operation would be instantaneous.

The benefits of this partitioning technique are significant. First, the new data is loaded with minimal resource utilization. The new data is loaded into an entirely separate table, and the index processing and constraint processing are applied only to the new partition. If the sales table was 50 GB and had 12 partitions, then a new month's worth of data contains approximately 4 GB. Only the new month's worth of data must be indexed. None of the indexes on the remaining 46 GB of data must be modified at all. This partitioning scheme additionally ensures that the load processing time is directly proportional to the amount of new data being loaded, not to the total size of the sales table.

Second, the new data is loaded with minimal impact on concurrent queries. All of the operations associated with data loading are occurring on a separate `sales_01_2001` table. Therefore, none of the existing data or indexes of the `sales` table is affected during this data refresh process. The `sales` table and its indexes remain entirely untouched throughout this refresh process.

Third, in case of the existence of any global indexes, those are incrementally maintained as part of the exchange command. This maintenance does not affect the availability of the existing global index structures.

The exchange operation can be viewed as a publishing mechanism. Until the data warehouse administrator exchanges the `sales_01_2001` table into the `sales` table, end users cannot see the new data. Once the exchange has occurred, then any end user query accessing the `sales` table will immediately be able to see the `sales_01_2001` data.

Partitioning is useful not only for adding new data but also for removing and archiving data. Many data warehouses maintain a rolling window of data. For example, the data warehouse stores the most recent 36 months of `sales` data. Just as a new partition can be added to the `sales` table (as described earlier), an old partition can be quickly (and independently) removed from the `sales` table. These two benefits (reduced resources utilization and minimal end-user impact) are just as pertinent to removing a partition as they are to adding a partition.

Removing data from a partitioned table does not necessarily mean that the old data is physically deleted from the database. There are two alternatives for removing old data from a partitioned table. First, you can physically delete all data from the database by dropping the partition containing the old data, thus freeing the allocated space:

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
```

Also, you can exchange the old partition with an empty table of the same structure; this empty table is created equivalent to steps 1 and 2 described in the load process. Assuming the new empty table stub is named `sales_archive_01_1998`, the following SQL statement will empty partition `sales_01_1998`:

```
ALTER TABLE sales EXCHANGE PARTITION sales_01_1998
WITH TABLE sales_archive_01_1998 INCLUDING INDEXES WITHOUT VALIDATION
UPDATE GLOBAL INDEXES;
```

Note that the old data is still existent as the exchanged, nonpartitioned table `sales_archive_01_1998`.

If the partitioned table was setup in a way that every partition is stored in a separate tablespace, you can archive (or transport) this table using Oracle Database's transportable tablespace framework before dropping the actual data (the tablespace). See "Transportation Using Transportable Tablespaces" on page 16-4 for further details regarding transportable tablespaces.

In some situations, you might not want to drop the old data immediately, but keep it as part of the partitioned table; although the data is no longer of main interest, there are still potential queries accessing this old, read-only data. You can use Oracle's data compression to minimize the space usage of the old data. We also assume that at least one compressed partition is already part of the partitioned table.

See Also:

- Chapter 3, "Physical Design in Data Warehouses" for a generic discussion of table compression
- *Oracle Database VLDB and Partitioning Guide* for more information regarding partitioning and table compression

Refresh Scenarios

A typical scenario might not only need to compress old data, but also to merge several old partitions to reflect the granularity for a later backup of several merged partitions. Let us assume that a backup (partition) granularity is on a quarterly base for any quarter, where the oldest month is more than 36 months behind the most recent month. In this case, we are therefore compressing and merging `sales_01_1998`, `sales_02_1998`, and `sales_03_1998` into a new, compressed partition `sales_q1_1998`.

1. Create the new merged partition in parallel in another tablespace. The partition will be compressed as part of the `MERGE` operation:

```
ALTER TABLE sales MERGE PARTITIONS sales_01_1998, sales_02_1998, sales_03_1998
 INTO PARTITION sales_q1_1998 TABLESPACE archive_q1_1998
COMPRESS UPDATE GLOBAL INDEXES PARALLEL 4;
```

2. The partition `MERGE` operation invalidates the local indexes for the new merged partition. We therefore have to rebuild them:

```
ALTER TABLE sales MODIFY PARTITION sales_q1_1998
REBUILD UNUSABLE LOCAL INDEXES;
```

Alternatively, you can choose to create the new compressed table outside the partitioned table and exchange it back. The performance and the temporary space consumption is identical for both methods:

1. Create an intermediate table to hold the new merged information. The following statement inherits all `NOT NULL` constraints from the original table by default:

```
CREATE TABLE sales_q1_1998_out TABLESPACE archive_q1_1998
NOLOGGING COMPRESS PARALLEL 4 AS SELECT * FROM sales
WHERE time_id >=  TO_DATE('01-JAN-1998','dd-mon-yyyy')
  AND time_id < TO_DATE('01-APR-1998','dd-mon-yyyy');
```

2. Create the equivalent index structure for table `sales_q1_1998_out` than for the existing table `sales`.
3. Prepare the existing table sales for the exchange with the new compressed table `sales_q1_1998_out`. Because the table to be exchanged contains data actually covered in three partitions, we have to create one matching partition, having the range boundaries we are looking for. You simply have to drop two of the existing partitions. Note that you have to drop the lower two partitions `sales_01_1998` and `sales_02_1998`; the lower boundary of a range partition is always defined by the upper (exclusive) boundary of the previous partition:

```
ALTER TABLE sales DROP PARTITION sales_01_1998;
ALTER TABLE sales DROP PARTITION sales_02_1998;
```

4. You can now exchange table `sales_q1_1998_out` with partition `sales_03_1998`. Unlike what the name of the partition suggests, its boundaries cover Q1-1998.

```
ALTER TABLE sales EXCHANGE PARTITION sales_03_1998
```

```
WITH TABLE sales_q1_1998_out INCLUDING INDEXES WITHOUT VALIDATION
UPDATE GLOBAL INDEXES;
```

Both methods apply to slightly different business scenarios: Using the `MERGE PARTITION` approach invalidates the local index structures for the affected partition, but it keeps all data accessible all the time. Any attempt to access the affected partition through one of the unusable index structures raises an error. The limited availability time is approximately the time for re-creating the local bitmap index structures. In most cases, this can be neglected, because this part of the partitioned table should not be accessed too often.

The CTAS approach, however, minimizes unavailability of any index structures close to zero, but there is a specific time window, where the partitioned table does not have all the data, because we dropped two partitions. The limited availability time is approximately the time for exchanging the table. Depending on the existence and number of global indexes, this time window varies. Without any existing global indexes, this time window is a matter of a fraction to few seconds.

These examples are a simplification of the data warehouse rolling window load scenario. Real-world data warehouse refresh characteristics are always more complex. However, the advantages of this rolling window approach are not diminished in more complex scenarios.

Note that before you add single or multiple compressed partitions to a partitioned table for the first time, all local bitmap indexes must be either dropped or marked unusable. After the first compressed partition is added, no additional actions are necessary for all subsequent operations involving compressed partitions. It is irrelevant how the compressed partitions are added to the partitioned table.

See Also:

- *Oracle Database VLDB and Partitioning Guide* for more information regarding partitioning and table compression
- *Oracle Database Administrator's Guide* for further details about partitioning and table compression.

Scenarios for Using Partitioning for Refreshing Data Warehouses

This section contains two typical scenarios where partitioning is used with refresh.

Refresh Scenario 1

Data is loaded daily. However, the data warehouse contains two years of data, so that partitioning by day might not be desired.

The solution is to partition by week or month (as appropriate). Use `INSERT` to add the new data to an existing partition. The `INSERT` operation only affects a single partition, so the benefits described previously remain intact. The `INSERT` operation could occur while the partition remains a part of the table. Inserts into a single partition can be parallelized:

```
INSERT /*+ APPEND*/ INTO sales PARTITION (sales_01_2001)
SELECT * FROM new_sales;
```

The indexes of this `sales` partition will be maintained in parallel as well. An alternative is to use the `EXCHANGE` operation. You can do this by exchanging the `sales_01_2001` partition of the `sales` table and then using an `INSERT` operation. You might prefer this technique when dropping and rebuilding indexes is more efficient than maintaining them.

Refresh Scenario 2

New data feeds, although consisting primarily of data for the most recent day, week, and month, also contain some data from previous time periods.

Solution 1 Use parallel SQL operations (such as `CREATE TABLE ... AS SELECT`) to separate the new data from the data in previous time periods. Process the old data separately using other techniques.

New data feeds are not solely time based. You can also feed new data into a data warehouse with data from multiple operational systems on a business need basis. For example, the sales data from direct channels may come into the data warehouse separately from the data from indirect channels. For business reasons, it may furthermore make sense to keep the direct and indirect data in separate partitions.

Solution 2 Oracle supports composite range-list partitioning. The primary partitioning strategy of the sales table could be range partitioning based on `time_id` as shown in the example. However, the subpartitioning is a list based on the channel attribute. Each subpartition can now be loaded independently of each other (for each distinct channel) and added in a rolling window operation as discussed before. The partitioning strategy addresses the business needs in the most optimal manner.

Optimizing DML Operations During Refresh

You can optimize DML performance through the following techniques:

- Implementing an Efficient MERGE Operation
- Maintaining Referential Integrity
- Purging Data

Implementing an Efficient MERGE Operation

Commonly, the data that is extracted from a source system is not simply a list of new records that needs to be inserted into the data warehouse. Instead, this new data set is a combination of new records as well as modified records. For example, suppose that most of data extracted from the OLTP systems will be new sales transactions. These records are inserted into the warehouse's `sales` table, but some records may reflect modifications of previous transactions, such as returned merchandise or transactions that were incomplete or incorrect when initially loaded into the data warehouse. These records require updates to the `sales` table.

As a typical scenario, suppose that there is a table called `new_sales` that contains both inserts and updates that are applied to the `sales` table. When designing the entire data warehouse load process, it was determined that the `new_sales` table would contain records with the following semantics:

- If a given `sales_transaction_id` of a record in `new_sales` already exists in `sales`, then update the `sales` table by adding the `sales_dollar_amount` and `sales_quantity_sold` values from the `new_sales` table to the existing row in the `sales` table.
- Otherwise, insert the entire new record from the `new_sales` table into the `sales` table.

This `UPDATE-ELSE-INSERT` operation is often called a merge. A merge can be executed using one SQL statement.

Example 16–1 MERGE Operation

```
MERGE INTO sales s USING new_sales n
ON (s.sales_transaction_id = n.sales_transaction_id)
WHEN MATCHED THEN
UPDATE SET s.sales_quantity_sold = s.sales_quantity_sold + n.sales_quantity_sold,
 s.sales_dollar_amount = s.sales_dollar_amount + n.sales_dollar_amount
WHEN NOT MATCHED THEN INSERT (sales_transaction_id, sales_quantity_sold,
sales_dollar_amount)
VALUES (n.sales_transcation_id, n.sales_quantity_sold, n.sales_dollar_amount);
```

In addition to using the MERGE statement for unconditional UPDATE ELSE INSERT functionality into a target table, you can also use it to:

- Perform an UPDATE only or INSERT only statement.
- Apply additional WHERE conditions for the UPDATE or INSERT portion of the MERGE statement.
- The UPDATE operation can even delete rows if a specific condition yields true.

Example 16–2 Omitting the INSERT Clause

In some data warehouse applications, it is not allowed to add new rows to historical information, but only to update them. It may also happen that you do not want to update but only insert new information. The following example demonstrates INSERT-only with UPDATE-only functionality:

```
MERGE USING Product_Changes S     -- Source/Delta table
INTO Products D1                  -- Destination table 1
ON (D1.PROD_ID = S.PROD_ID)       -- Search/Join condition
WHEN MATCHED THEN UPDATE          -- update if join
SET D1.PROD_STATUS = S.PROD_NEW_STATUS
```

Example 16–3 Omitting the UPDATE Clause

The following statement illustrates an example of omitting an UPDATE:

```
MERGE USING New_Product S           -- Source/Delta table
INTO Products D2                    -- Destination table 2
ON (D2.PROD_ID = S.PROD_ID)         -- Search/Join condition
WHEN NOT MATCHED THEN               -- insert if no join
INSERT (PROD_ID, PROD_STATUS) VALUES (S.PROD_ID, S.PROD_NEW_STATUS)
```

When the INSERT clause is omitted, Oracle performs a regular join of the source and the target tables. When the UPDATE clause is omitted, Oracle performs an antijoin of the source and the target tables. This makes the join between the source and target table more efficient.

Example 16–4 Skipping the UPDATE Clause

In some situations, you may want to skip the UPDATE operation when merging a given row into the table. In this case, you can use an optional WHERE clause in the UPDATE clause of the MERGE. As a result, the UPDATE operation only executes when a given condition is true. The following statement illustrates an example of skipping the UPDATE operation:

```
MERGE
USING Product_Changes S                     -- Source/Delta table
INTO Products P                             -- Destination table 1
ON (P.PROD_ID = S.PROD_ID)                  -- Search/Join condition
WHEN MATCHED THEN
```

```
UPDATE                                          -- update if join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE P.PROD_STATUS <> "OBSOLETE"               -- Conditional UPDATE
```

This shows how the UPDATE operation would be skipped if the condition P.PROD_STATUS <> "OBSOLETE" is not true. The condition predicate can refer to both the target and the source table.

Example 16–5 Conditional Inserts with MERGE Statements

You may want to skip the INSERT operation when merging a given row into the table. So an optional WHERE clause is added to the INSERT clause of the MERGE. As a result, the INSERT operation only executes when a given condition is true. The following statement offers an example:

```
MERGE USING Product_Changes S                       -- Source/Delta table
INTO Products P                                     -- Destination table 1
ON (P.PROD_ID = S.PROD_ID)                          -- Search/Join condition
WHEN MATCHED THEN UPDATE                            -- update if join
SET P.PROD_LIST_PRICE = S.PROD_NEW_PRICE
WHERE P.PROD_STATUS <> "OBSOLETE"                   -- Conditional
WHEN NOT MATCHED THEN
INSERT (PROD_ID, PROD_STATUS, PROD_LIST_PRICE)      -- insert if not join
VALUES (S.PROD_ID, S.PROD_NEW_STATUS, S.PROD_NEW_PRICE)
WHERE S.PROD_STATUS <> "OBSOLETE";                  -- Conditional INSERT
```

This example shows that the INSERT operation would be skipped if the condition S.PROD_STATUS <> "OBSOLETE" is not true, and INSERT will only occur if the condition is true. The condition predicate can refer to the source table only. The condition predicate can only refer to the source table.

Example 16–6 Using the DELETE Clause with MERGE Statements

You may want to cleanse tables while populating or updating them. To do this, you may want to consider using the DELETE clause in a MERGE statement, as in the following example:

```
MERGE USING Product_Changes S
INTO Products D ON (D.PROD_ID = S.PROD_ID)
WHEN MATCHED THEN
UPDATE SET D.PROD_LIST_PRICE =S.PROD_NEW_PRICE, D.PROD_STATUS = S.PROD_NEWSTATUS
DELETE WHERE (D.PROD_STATUS = "OBSOLETE")
WHEN NOT MATCHED THEN
INSERT (PROD_ID, PROD_LIST_PRICE, PROD_STATUS)
VALUES (S.PROD_ID, S.PROD_NEW_PRICE, S.PROD_NEW_STATUS);
```

Thus when a row is updated in products, Oracle checks the delete condition D.PROD_STATUS = "OBSOLETE", and deletes the row if the condition yields true.

The DELETE operation is not as same as that of a complete DELETE statement. Only the rows from the destination of the MERGE can be deleted. The only rows that are affected by the DELETE are the ones that are updated by this MERGE statement. Thus, although a given row of the destination table meets the delete condition, if it does not join under the ON clause condition, it is not deleted.

Example 16–7 Unconditional Inserts with MERGE Statements

You may want to insert all of the source rows into a table. In this case, the join between the source and target table can be avoided. By identifying special constant join

conditions that always result to FALSE, for example, 1=0, such MERGE statements will be optimized and the join condition will be suppressed.

```
MERGE USING New_Product S       -- Source/Delta table
INTO Products P                 -- Destination table 1
ON (1 = 0)                      -- Search/Join condition
WHEN NOT MATCHED THEN           -- insert if no join
INSERT (PROD_ID, PROD_STATUS) VALUES (S.PROD_ID, S.PROD_NEW_STATUS)
```

Maintaining Referential Integrity

In some data warehousing environments, you might want to insert new data into tables in order to guarantee referential integrity. For example, a data warehouse may derive sales from an operational system that retrieves data directly from cash registers. sales is refreshed nightly. However, the data for the product dimension table may be derived from a separate operational system. The product dimension table may only be refreshed once for each week, because the product table changes relatively slowly. If a new product was introduced on Monday, then it is possible for that product's product_id to appear in the sales data of the data warehouse before that product_id has been inserted into the data warehouses product table.

Although the sales transactions of the new product may be valid, this sales data will not satisfy the referential integrity constraint between the product dimension table and the sales fact table. Rather than disallow the new sales transactions, you might choose to insert the sales transactions into the sales table. However, you might also wish to maintain the referential integrity relationship between the sales and product tables. This can be accomplished by inserting new rows into the product table as placeholders for the unknown products.

As in previous examples, we assume that the new data for the sales table is staged in a separate table, new_sales. Using a single INSERT statement (which can be parallelized), the product table can be altered to reflect the new products:

```
INSERT INTO product
  (SELECT sales_product_id, 'Unknown Product Name', NULL, NULL ...
   FROM new_sales WHERE sales_product_id NOT IN
  (SELECT product_id FROM product));
```

Purging Data

Occasionally, it is necessary to remove large amounts of data from a data warehouse. A very common scenario is the rolling window discussed previously, in which older data is rolled out of the data warehouse to make room for new data.

However, sometimes other data might need to be removed from a data warehouse. Suppose that a retail company has previously sold products from XYZ Software, and that XYZ Software has subsequently gone out of business. The business users of the warehouse may decide that they are no longer interested in seeing any data related to XYZ Software, so this data should be deleted.

One approach to removing a large volume of data is to use parallel delete as shown in the following statement:

```
DELETE FROM sales WHERE sales_product_id IN (SELECT product_id
   FROM product WHERE product_category = 'XYZ Software');
```

This SQL statement spawns one parallel process for each partition. This approach is much more efficient than a serial DELETE statement, and none of the data in the sales table will need to be moved. However, this approach also has some disadvantages. When removing a large percentage of rows, the DELETE statement will

leave many empty row-slots in the existing partitions. If new data is being loaded using a rolling window technique (or is being loaded using direct-path INSERT or load), then this storage space will not be reclaimed. Moreover, even though the DELETE statement is parallelized, there might be more efficient methods. An alternative method is to re-create the entire sales table, keeping the data for all product categories except XYZ Software.

```
CREATE TABLE sales2 AS SELECT * FROM sales, product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'XYZ Software'
NOLOGGING PARALLEL (DEGREE 8)
#PARTITION ... ; #create indexes, constraints, and so on
DROP TABLE SALES;
RENAME SALES2 TO SALES;
```

This approach may be more efficient than a parallel delete. However, it is also costly in terms of the amount of disk space, because the sales table must effectively be instantiated twice.

An alternative method to utilize less space is to re-create the sales table one partition at a time:

```
CREATE TABLE sales_temp AS SELECT * FROM sales WHERE 1=0;
INSERT INTO sales_temp
SELECT * FROM sales PARTITION (sales_99jan), product
WHERE sales.sales_product_id = product.product_id
AND product_category <> 'XYZ Software';
<create appropriate indexes and constraints on sales_temp>
ALTER TABLE sales EXCHANGE PARTITION sales_99jan WITH TABLE sales_temp;
```

Continue this process for each partition in the sales table.

Refreshing Materialized Views

When creating a materialized view, you have the option of specifying whether the refresh occurs ON DEMAND or ON COMMIT. In the case of ON COMMIT, the materialized view is changed every time a transaction commits, thus ensuring that the materialized view always contains the latest data. Alternatively, you can control the time when refresh of the materialized views occurs by specifying ON DEMAND. In this case, the materialized view can only be refreshed by calling one of the procedures in the DBMS_MVIEW package.

DBMS_MVIEW provides three different types of refresh operations.

- DBMS_MVIEW.REFRESH

  Refresh one or more materialized views.

- DBMS_MVIEW.REFRESH_ALL_MVIEWS

  Refresh all materialized views.

- DBMS_MVIEW.REFRESH_DEPENDENT

  Refresh all materialized views that depend on a specified master table or materialized view or list of master tables or materialized views.

  See Also: "Manual Refresh Using the DBMS_MVIEW Package" on page 16-12 for more information

Performing a refresh operation requires temporary space to rebuild the indexes and can require additional space for performing the refresh operation itself. Some sites might prefer not to refresh all of their materialized views at the same time: as soon as some underlying detail data has been updated, all materialized views using this data will become stale. Therefore, if you defer refreshing your materialized views, you can either rely on your chosen rewrite integrity level to determine whether or not a stale materialized view can be used for query rewrite, or you can temporarily disable query rewrite with an `ALTER SYSTEM SET QUERY_REWRITE_ENABLED = FALSE` statement. After refreshing the materialized views, you can re-enable query rewrite as the default for all sessions in the current database instance by specifying `ALTER SYSTEM SET QUERY_REWRITE_ENABLED` as `TRUE`. Refreshing a materialized view automatically updates all of its indexes. In the case of full refresh, this requires temporary sort space to rebuild all indexes during refresh. This is because the full refresh truncates or deletes the table before inserting the new full data volume. If insufficient temporary space is available to rebuild the indexes, then you must explicitly drop each index or mark it `UNUSABLE` prior to performing the refresh operation.

If you anticipate performing insert, update or delete operations on tables referenced by a materialized view concurrently with the refresh of that materialized view, and that materialized view includes joins and aggregation, Oracle recommends you use `ON COMMIT` fast refresh rather than `ON DEMAND` fast refresh.

> **See Also:** *Oracle OLAP User's Guide* for information regarding the refresh of cube organized materialized views

Complete Refresh

A complete refresh occurs when the materialized view is initially defined as `BUILD IMMEDIATE`, unless the materialized view references a prebuilt table. For materialized views using `BUILD DEFERRED`, a complete refresh must be requested before it can be used for the first time. A complete refresh may be requested at any time during the life of any materialized view. The refresh involves reading the detail tables to compute the results for the materialized view. This can be a very time-consuming process, especially if there are huge amounts of data to be read and processed. Therefore, you should always consider the time required to process a complete refresh before requesting it.

There are, however, cases when the only refresh method available for an already built materialized view is complete refresh because the materialized view does not satisfy the conditions specified in the following section for a fast refresh.

Fast Refresh

Most data warehouses have periodic incremental updates to their detail data. As described in "Materialized View Schema Design" on page 9-5, you can use the SQL*Loader or any bulk load utility to perform incremental loads of detail data. Fast refresh of your materialized views is usually efficient, because instead of having to recompute the entire materialized view, the changes are applied to the existing data. Thus, processing only the changes can result in a very fast refresh time.

Partition Change Tracking (PCT) Refresh

When there have been some partition maintenance operations on the detail tables, this is the only method of fast refresh that can be used. PCT-based refresh on a materialized view is enabled only if all the conditions described in "Partition Change Tracking" on page 10-1 are satisfied.

In the absence of partition maintenance operations on detail tables, when you request a FAST method (method => 'F') of refresh through procedures in DBMS_MVIEW package, Oracle uses a heuristic rule to try log-based rule fast refresh before choosing PCT refresh. Similarly, when you request a FORCE method (method => '?'), Oracle chooses the refresh method based on the following attempt order: log-based fast refresh, PCT refresh, and complete refresh. Alternatively, you can request the PCT method (method => 'P'), and Oracle uses the PCT method provided all PCT requirements are satisfied.

Oracle can use TRUNCATE PARTITION on a materialized view if it satisfies the conditions in "Benefits of Partitioning a Materialized View" on page 10-6 and hence, make the PCT refresh process more efficient.

ON COMMIT Refresh

A materialized view can be refreshed automatically using the ON COMMIT method. Therefore, whenever a transaction commits which has updated the tables on which a materialized view is defined, those changes are automatically reflected in the materialized view. The advantage of using this approach is you never have to remember to refresh the materialized view. The only disadvantage is the time required to complete the commit will be slightly longer because of the extra processing involved. However, in a data warehouse, this should not be an issue because there is unlikely to be concurrent processes trying to update the same table.

Manual Refresh Using the DBMS_MVIEW Package

When a materialized view is refreshed ON DEMAND, one of four refresh methods can be specified as shown in the following table. You can define a default option during the creation of the materialized view. Table 16–1 details the refresh options.

Table 16–1 ON DEMAND Refresh Methods

Refresh Option	Parameter	Description
COMPLETE	C	Refreshes by recalculating the defining query of the materialized view.
FAST	F	Refreshes by incrementally applying changes to the materialized view. For local materialized views, it chooses the refresh method which is estimated by optimizer to be most efficient. The refresh methods considered are log-based FAST and FAST_PCT.
FAST_PCT	P	Refreshes by recomputing the rows in the materialized view affected by changed partitions in the detail tables.
FORCE	?	Attempts a fast refresh. If that is not possible, it does a complete refresh. For local materialized views, it chooses the refresh method which is estimated by optimizer to be most efficient. The refresh methods considered are log based FAST, FAST_PCT, and COMPLETE.

Three refresh procedures are available in the DBMS_MVIEW package for performing ON DEMAND refresh. Each has its own unique set of parameters.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the DBMS_MVIEW package
- *Oracle Database Advanced Replication* for information showing how to use it in a replication environment

Refresh Specific Materialized Views with REFRESH

Use the DBMS_MVIEW.REFRESH procedure to refresh one or more materialized views. Some parameters are used only for replication, so they are not mentioned here. The required parameters to use this procedure are:

- The comma-delimited list of materialized views to refresh
- The refresh method: F-Fast, P-Fast_PCT, ?-Force, C-Complete
- The rollback segment to use
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter is set to the number of refreshes that failed, and a generic error message indicates that failures occurred. The alert log for the instance gives details of refresh errors. If set to FALSE, the default, then refresh will stop after it encounters the first error, and any remaining materialized views in the list are not be refreshed.
- The following four parameters are used by the replication process. For warehouse refresh, set them to FALSE, 0,0,0.
- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then the refresh of each specified materialized view is done in a separate transaction. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views. When a materialized view is refreshed in atomic mode, it is eligible for query rewrite if the rewrite integrity mode is set to stale_tolerated. Atomic refresh cannot be guaranteed when refresh is performed on nested views.

For example, to perform a fast refresh on the materialized view cal_month_sales_mv, the DBMS_MVIEW package would be called as follows:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV', 'F', '', TRUE, FALSE, 0,0,0, FALSE);
```

Multiple materialized views can be refreshed at the same time, and they do not all have to use the same refresh method. To give them different refresh methods, specify multiple method codes in the same order as the list of materialized views (without commas). For example, the following specifies that cal_month_sales_mv be completely refreshed and fweek_pscat_sales_mv receive a fast refresh:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV, FWEEK_PSCAT_SALES_MV', 'CF', '',
  TRUE, FALSE, 0,0,0, FALSE);
```

If the refresh method is not specified, the default refresh method as specified in the materialized view definition is used.

Refresh All Materialized Views with REFRESH_ALL_MVIEWS

An alternative to specifying the materialized views to refresh is to use the procedure DBMS_MVIEW.REFRESH_ALL_MVIEWS. This procedure refreshes all materialized views. If any of the materialized views fails to refresh, then the number of failures is reported.

The parameters for this procedure are:

- The number of failures (this is an OUT variable)
- The refresh method: F-Fast, P-Fast_PCT, ?-Force, C-Complete

- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter is set to the number of refreshes that failed, and a generic error message indicates that failures occurred. The alert log for the instance gives details of refresh errors. If set to FALSE, the default, then refresh stops after it encounters the first error, and any remaining materialized views in the list is not refreshed.

- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then each of the materialized views is refreshed non-atomically in separate transactions. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views. When a materialized view is refreshed in atomic mode, it is eligible for query rewrite if the rewrite integrity mode is set to stale_tolerated. Atomic refresh cannot be guaranteed when refresh is performed on nested views.

An example of refreshing all materialized views is the following:

```
DBMS_MVIEW.REFRESH_ALL_MVIEWS(failures,'C','', TRUE, FALSE);
```

Refresh Dependent Materialized Views with REFRESH_DEPENDENT

The third procedure, DBMS_MVIEW.REFRESH_DEPENDENT, refreshes only those materialized views that depend on a specific table or list of tables. For example, suppose the changes have been received for the orders table but not for customer payments. The refresh dependent procedure can be called to refresh only those materialized views that reference the orders table.

The parameters for this procedure are:

- The number of failures (this is an OUT variable)
- The dependent table
- The refresh method: F-Fast, P-Fast_PCT, ?-Force, C-Complete
- The rollback segment to use
- Refresh after errors (TRUE or FALSE)

  A Boolean parameter. If set to TRUE, the number_of_failures output parameter is set to the number of refreshes that failed, and a generic error message indicates that failures occurred. The alert log for the instance gives details of refresh errors. If set to FALSE, the default, then refresh stops after it encounters the first error, and any remaining materialized views in the list are not refreshed.

- Atomic refresh (TRUE or FALSE)

  If set to TRUE, then all refreshes are done in one transaction. If set to FALSE, then the refresh of each specified materialized view is done in a separate transaction. If set to FALSE, Oracle can optimize refresh by using parallel DML and truncate DDL on a materialized views. When a materialized view is refreshed in atomic mode, it is eligible for query rewrite if the rewrite integrity mode is set to stale_tolerated. Atomic refresh cannot be guaranteed when refresh is performed on nested views.

- Whether it is nested or not

  If set to TRUE, refresh all the dependent materialized views of the specified set of tables based on a dependency order to ensure the materialized views are truly fresh with respect to the underlying base tables.

To perform a full refresh on all materialized views that reference the customers table, specify:

```
DBMS_MVIEW.REFRESH_DEPENDENT(failures, 'CUSTOMERS', 'C', '', FALSE, FALSE );
```

Using Job Queues for Refresh

Job queues can be used to refresh multiple materialized views in parallel. If queues are not available, fast refresh sequentially refreshes each view in the foreground process. To make queues available, you must set the JOB_QUEUE_PROCESSES parameter. This parameter defines the number of background job queue processes and determines how many materialized views can be refreshed concurrently. Oracle tries to balance the number of concurrent refreshes with the degree of parallelism of each refresh. The order in which the materialized views are refreshed is determined by dependencies imposed by nested materialized views and potential for efficient refresh by using query rewrite against other materialized views (See "Scheduling Refresh" on page 16-18 for details). This parameter is only effective when atomic_refresh is set to FALSE.

If the process that is executing DBMS_MVIEW.REFRESH is interrupted or the instance is shut down, any refresh jobs that were executing in job queue processes will be requeued and will continue running. To remove these jobs, use the DBMS_JOB.REMOVE procedure.

When Fast Refresh is Possible

Not all materialized views may be fast refreshable. Therefore, use the package DBMS_MVIEW.EXPLAIN_MVIEW to determine what refresh methods are available for a materialized view. See Chapter 9, "Basic Materialized Views" for further information about the DBMS_MVIEW package.

If you are not sure how to make a materialized view fast refreshable, you can use the DBMS_ADVISOR.TUNE_MVIEW procedure, which will provide a script containing the statements required to create a fast refreshable materialized view. See the *Oracle Database Performance Tuning Guide* for more information.

Recommended Initialization Parameters for Parallelism

The following initialization parameters need to be set properly for parallelism to be effective:

- PARALLEL_MAX_SERVERS should be set high enough to take care of parallelism. You must consider the number of slaves needed for the refresh statement. For example, with a degree of parallelism of eight, you need 16 slave processes.
- PGA_AGGREGATE_TARGET should be set for the instance to manage the memory usage for sorts and joins automatically. If the memory parameters are set manually, SORT_AREA_SIZE should be less than HASH_AREA_SIZE.
- OPTIMIZER_MODE should equal all_rows.

Remember to analyze all tables and indexes for better optimization.

> **See Also:** *Oracle Database VLDB and Partitioning Guide* for further information

Monitoring a Refresh

While a job is running, you can query the V$SESSION_LONGOPS view to tell you the progress of each materialized view being refreshed.

```
SELECT * FROM V$SESSION_LONGOPS;
```

To look at the progress of which jobs are on which queue, use:

```
SELECT * FROM DBA_JOBS_RUNNING;
```

Checking the Status of a Materialized View

Three views are provided for checking the status of a materialized view: DBA_MVEIWS, ALL_MVIEWS, and USER_MVIEWS. To check if a materialized view is fresh or stale, issue the following statement:

```
SELECT MVIEW_NAME, STALENESS, LAST_REFRESH_TYPE, COMPILE_STATE
FROM USER_MVIEWS ORDER BY MVIEW_NAME;

MVIEW_NAME            STALENESS      LAST_REF       COMPILE_STATE
----------            ---------      --------       -------------
CUST_MTH_SALES_MV     NEEDS_COMPILE  FAST           NEEDS_COMPILE
PROD_YR_SALES_MV      FRESH          FAST           VALID
```

If the compile_state column shows NEEDS COMPILE, the other displayed column values cannot be trusted as reflecting the true status. To revalidate the materialized view, issue the following statement:

```
ALTER MATERIALIZED VIEW [materialized_view_name] COMPILE;
```

Then reissue the SELECT statement.

Viewing Partition Freshness

Several views are available that enable you to verify the status of base table partitions and determine which ranges of materialized view data are fresh and which are stale. The views are as follows:

- *_USER_MVIEWS

  To determine Partition Change Tracking (PCT) information for the materialized view.

- *_USER_MVIEW_DETAIL_RELATIONS

  To display partition information for the detail table a materialized view is based on.

- *_USER_MVIEW_DETAIL_PARTITION

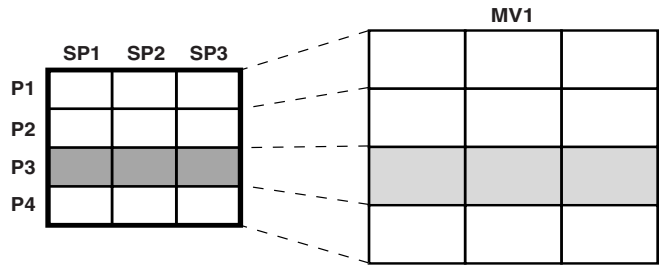
  To determine which partitions are fresh.

- *_USER_MVIEW_DETAIL_SUBPARTITION

  To determine which subpartitions are fresh.

The use of these views is illustrated in the following examples. Figure 16–1 illustrates a range-list partitioned table and a materialized view based on it. The partitions are P1, P2, P3, and P4, while the subpartitions are SP1, SP2, and SP3.

Figure 16–1 Determining PCT Freshness



Examples of Using Views to Determine Freshness This section illustrates examples of determining the PCT and freshness information for materialized views and their detail tables.

Example 16–8 Verifying the PCT Status of a Materialized View

Query USER_MVIEWS to access PCT information about the materialized view, as shown in the following:

```
SELECT MVIEW_NAME, NUM_PCT_TABLES, NUM_FRESH_PCT_REGIONS,
   NUM_STALE_PCT_REGIONS
FROM USER_MVIEWS
WHERE MVIEW_NAME = MV1;

MVIEW_NAME NUM_PCT_TABLES NUM_FRESH_PCT_REGIONS NUM_STALE_PCT_REGIONS
---------- -------------- --------------------- ---------------------
       MV1              1                     9                     3
```

Example 16–9 Verifying the PCT Status in a Materialized View's Detail Table

Query USER_MVIEW_DETAIL_RELATIONS to access PCT detail table information, as shown in the following:

```
SELECT MVIEW_NAME, DETAILOBJ_NAME, DETAILOBJ_PCT,
   NUM_FRESH_PCT_PARTITIONS, NUM_STALE_PCT_PARTITIONS
FROM USER_MVIEW_DETAIL_RELATIONS
WHERE MVIEW_NAME = MV1;

MVIEW_NAME  DETAILOBJ_NAME  DETAIL_OBJ_PCT  NUM_FRESH_PCT_PARTITIONS  NUM_STALE_PCT_PARTITIONS
----------  --------------  --------------  ------------------------  ------------------------
        MV1             T1               Y                         3                         1
```

Example 16–10 Verifying Which Partitions are Fresh

Query USER_MVIEW_DETAIL_PARTITION to access PCT freshness information for partitions, as shown in the following:

```
SELECT MVIEW_NAME,DETAILOBJ_NAME,DETAIL_PARTITION_NAME,
   DETAIL_PARTITION_POSITION,FRESHNESS
FROM USER_MVIEW_DETAIL_PARTITION
WHERE MVIEW_NAME = MV1;

MVIEW_NAME  DETAILOBJ_NAME  DETAIL_PARTITION_NAME  DETAIL_PARTITION_POSITION  FRESHNESS
----------  --------------  ---------------------  -------------------------  ---------
       MV1              T1                     P1                          1      FRESH
       MV1              T1                     P2                          2      FRESH
       MV1              T1                     P3                          3      STALE
       MV1              T1                     P4                          4      FRESH
```

Example 16–11 Verifying Which Subpartitions are Fresh

Query USER_MVIEW_DETAIL_SUBPARTITION to access PCT freshness information for subpartitions, as shown in the following:

```
SELECT MVIEW_NAME,DETAILOBJ_NAME,DETAIL_PARTITION_NAME, DETAIL_SUBPARTITION_NAME,
    DETAIL_SUBPARTITION_POSITION,FRESHNESS
FROM USER_MVIEW_DETAIL_SUBPARTITION
WHERE MVIEW_NAME = MV1;
```

```
MVIEW_NAME DETAILOBJ DETAIL_PARTITION DETAIL_SUBPARTITION_NAME DETAIL_SUBPARTITION_POS FRESHNESS
---------- --------- ---------------- ------------------------ ------------------------ ---------
       MV1        T1               P1                       SP1                        1     FRESH
       MV1        T1               P1                       SP2                        1     FRESH
       MV1        T1               P1                       SP3                        1     FRESH
       MV1        T1               P2                       SP1                        1     FRESH
       MV1        T1               P2                       SP2                        1     FRESH
       MV1        T1               P2                       SP3                        1     FRESH
       MV1        T1               P3                       SP1                        1     STALE
       MV1        T1               P3                       SP2                        1     STALE
       MV1        T1               P3                       SP3                        1     STALE
       MV1        T1               P4                       SP1                        1     FRESH
       MV1        T1               P4                       SP2                        1     FRESH
       MV1        T1               P4                       SP3                        1     FRESH
```

Scheduling Refresh

Very often you have multiple materialized views in the database. Some of these can be computed by rewriting against others. This is very common in data warehousing environment where you may have nested materialized views or materialized views at different levels of some hierarchy.

In such cases, you should create the materialized views as BUILD DEFERRED, and then issue one of the refresh procedures in DBMS_MVIEW package to refresh all the materialized views. Oracle Database computes the dependencies and refreshes the materialized views in the right order. Consider the example of a complete hierarchical cube described in "Examples of Hierarchical Cube Materialized Views" on page 21-24. Suppose all the materialized views have been created as BUILD DEFERRED. Creating the materialized views as BUILD DEFERRED only creates the metadata for all the materialized views. And, then, you can just call one of the refresh procedures in DBMS_MVIEW package to refresh all the materialized views in the right order:

```
DECLARE numerrs PLS_INTEGER;
BEGIN DBMS_MVIEW.REFRESH_DEPENDENT (
   number_of_failures => numerrs, list=>'SALES', method => 'C');
DBMS_OUTPUT.PUT_LINE('There were ' || numerrs || ' errors during refresh');
END;
/
```

The procedure refreshes the materialized views in the order of their dependencies (first sales_hierarchical_mon_cube_mv, followed by sales_hierarchical_qtr_cube_mv, then, sales_hierarchical_yr_cube_mv and finally, sales_hierarchical_all_cube_mv). Each of these materialized views gets rewritten against the one prior to it in the list).

The same kind of rewrite can also be used while doing PCT refresh. PCT refresh recomputes rows in a materialized view corresponding to changed rows in the detail tables. And, if there are other fresh materialized views available at the time of refresh, it can go directly against them as opposed to going against the detail tables.

Hence, it is always beneficial to pass a list of materialized views to any of the refresh procedures in DBMS_MVIEW package (irrespective of the method specified) and let the procedure figure out the order of doing refresh on materialized views.

Tips for Refreshing Materialized Views with Aggregates

Following are some guidelines for using the refresh mechanism for materialized views with aggregates.

- For fast refresh, create materialized view logs on all detail tables involved in a materialized view with the ROWID, SEQUENCE and INCLUDING NEW VALUES clauses.

  Include all columns from the table likely to be used in materialized views in the materialized view logs.

  Fast refresh may be possible even if the SEQUENCE option is omitted from the materialized view log. If it can be determined that only inserts or deletes will occur on all the detail tables, then the materialized view log does not require the SEQUENCE clause. However, if updates to multiple tables are likely or required or if the specific update scenarios are unknown, make sure the SEQUENCE clause is included.

- Use Oracle's bulk loader utility or direct-path INSERT (INSERT with the APPEND hint for loads).

  This is a lot more efficient than conventional insert. During loading, disable all constraints and re-enable when finished loading. Note that materialized view logs are required regardless of whether you use direct load or conventional DML.

  Try to optimize the sequence of conventional mixed DML operations, direct-path INSERT and the fast refresh of materialized views. You can use fast refresh with a mixture of conventional DML and direct loads. Fast refresh can perform significant optimizations if it finds that only direct loads have occurred, as illustrated in the following:

  1. Direct-path INSERT (SQL*Loader or INSERT /*+ APPEND */) into the detail table
  2. Refresh materialized view
  3. Conventional mixed DML
  4. Refresh materialized view

  You can use fast refresh with conventional mixed DML (INSERT, UPDATE, and DELETE) to the detail tables. However, fast refresh will be able to perform significant optimizations in its processing if it detects that only inserts or deletes have been done to the tables, such as:

  - DML INSERT or DELETE to the detail table
  - Refresh materialized views
  - DML update to the detail table
  - Refresh materialized view

  Even more optimal is the separation of INSERT and DELETE.

  If possible, refresh should be performed after each type of data change (as shown earlier) rather than issuing only one refresh at the end. If that is not possible, restrict the conventional DML to the table to inserts only, to get much better refresh performance. Avoid mixing deletes and direct loads.

Furthermore, for refresh ON COMMIT, Oracle keeps track of the type of DML done in the committed transaction. Therefore, do not perform direct-path INSERT and DML to other tables in the same transaction, as Oracle may not be able to optimize the refresh phase.

For ON COMMIT materialized views, where refreshes automatically occur at the end of each transaction, it may not be possible to isolate the DML statements, in which case keeping the transactions short will help. However, if you plan to make numerous modifications to the detail table, it may be better to perform them in one transaction, so that refresh of the materialized view will be performed just once at commit time rather than after each update.

- Oracle recommends partitioning the tables because it enables you to use:
  - Parallel DML

    For large loads or refresh, enabling parallel DML helps shorten the length of time for the operation.
  - Partition Change Tracking (PCT) fast refresh

    You can refresh your materialized views fast after partition maintenance operations on the detail tables. "Partition Change Tracking" on page 10-1 for details on enabling PCT for materialized views.
- Partitioning the materialized view also helps refresh performance as refresh can update the materialized view using parallel DML. For example, assume that the detail tables and materialized view are partitioned and have a parallel clause. The following sequence would enable Oracle to parallelize the refresh of the materialized view.
  1. Bulk load into the detail table.
  2. Enable parallel DML with an ALTER SESSION ENABLE PARALLEL DML statement.
  3. Refresh the materialized view.
- For refresh using DBMS_MVIEW.REFRESH, set the parameter atomic_refresh to FALSE.
  - For COMPLETE refresh, this will TRUNCATE to delete existing rows in the materialized view, which is faster than a delete.
  - For PCT refresh, if the materialized view is partitioned appropriately, this will use TRUNCATE PARTITION to delete rows in the affected partitions of the materialized view, which is faster than a delete.
  - For FAST or FORCE refresh, if COMPLETE or PCT refresh is chosen, this will be able to use the TRUNCATE optimizations described earlier.
- When using DBMS_MVIEW.REFRESH with JOB_QUEUES, remember to set atomic to FALSE. Otherwise, JOB_QUEUES is not used. Set the number of job queue processes greater than the number of processors.

  If job queues are enabled and there are many materialized views to refresh, it is faster to refresh all of them in a single command than to call them individually.
- Use REFRESH FORCE to ensure refreshing a materialized view so that it can definitely be used for query rewrite. The best refresh method is chosen. If a fast refresh cannot be done, a complete refresh is performed.
- Refresh all the materialized views in a single procedure call. This gives Oracle an opportunity to schedule refresh of all the materialized views in the right order

taking into account dependencies imposed by nested materialized views and potential for efficient refresh by using query rewrite against other materialized views.

Tips for Refreshing Materialized Views Without Aggregates

If a materialized view contains joins but no aggregates, then having an index on each of the join column rowids in the detail table will enhance refresh performance greatly, because this type of materialized view tends to be much larger than materialized views containing aggregates. For example, consider the following materialized view:

```
CREATE MATERIALIZED VIEW detail_fact_mv BUILD IMMEDIATE AS
SELECT s.rowid "sales_rid", t.rowid "times_rid", c.rowid "cust_rid",
   c.cust_state_province, t.week_ending_day, s.amount_sold
FROM sales s, times t, customers c
WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id;
```

Indexes should be created on columns `sales_rid`, `times_rid` and `cust_rid`. Partitioning is highly recommended, as is enabling parallel DML in the session before invoking refresh, because it will greatly enhance refresh performance.

This type of materialized view can also be fast refreshed if DML is performed on the detail table. It is recommended that the same procedure be applied to this type of materialized view as for a single table aggregate. That is, perform one type of change (direct-path `INSERT` or DML) and then refresh the materialized view. This is because Oracle Database can perform significant optimizations if it detects that only one type of change has been done.

Also, Oracle recommends that the refresh be invoked after each table is loaded, rather than load all the tables and then perform the refresh.

For refresh `ON COMMIT`, Oracle keeps track of the type of DML done in the committed transaction. Oracle therefore recommends that you do not perform direct-path and conventional DML to other tables in the same transaction because Oracle may not be able to optimize the refresh phase. For example, the following is not recommended:

1. Direct load new data into the fact table
2. DML into the store table
3. Commit

Also, try not to mix different types of conventional DML statements if possible. This would again prevent using various optimizations during fast refresh. For example, try to avoid the following:

1. Insert into the fact table
2. Delete from the fact table
3. Commit

If many updates are needed, try to group them all into one transaction because refresh will be performed just once at commit time, rather than after each update.

In a data warehousing environment, assuming that the materialized view has a parallel clause, the following sequence of steps is recommended:

1. Bulk load into the fact table
2. Enable parallel DML
3. An `ALTER SESSION ENABLE PARALLEL DML` statement

4. Refresh the materialized view

Tips for Refreshing Nested Materialized Views

All underlying objects are treated as ordinary tables when refreshing materialized views. If the ON COMMIT refresh option is specified, then all the materialized views are refreshed in the appropriate order at commit time. In other words, Oracle builds a partially ordered set of materialized views and refreshes them such that, after the successful completion of the refresh, all the materialized views are fresh. The status of the materialized views can be checked by querying the appropriate USER_, DBA_, or ALL_MVIEWS view.

If any of the materialized views are defined as ON DEMAND refresh (irrespective of whether the refresh method is FAST, FORCE, or COMPLETE), you will need to refresh them in the correct order (taking into account the dependencies between the materialized views) because the nested materialized view will be refreshed with respect to the current contents of the other materialized views (whether fresh or not). This can be achieved by invoking the refresh procedure against the materialized view at the top of the nested hierarchy and specifying the nested parameter as TRUE.

If a refresh fails during commit time, the list of materialized views that has not been refreshed is written to the alert log, and you must manually refresh them along with all their dependent materialized views.

Use the same DBMS_MVIEW procedures on nested materialized views that you use on regular materialized views.

These procedures have the following behavior when used with nested materialized views:

- If REFRESH is applied to a materialized view my_mv that is built on other materialized views, then my_mv will be refreshed with respect to the current contents of the other materialized views (that is, the other materialized views will not be made fresh first) unless you specify nested => TRUE.
- If REFRESH_DEPENDENT is applied to materialized view my_mv, then only materialized views that directly depend on my_mv will be refreshed (that is, a materialized view that depends on a materialized view that depends on my_mv will not be refreshed) unless you specify nested => TRUE.
- If REFRESH_ALL_MVIEWS is used, the order in which the materialized views will be refreshed is guaranteed to respect the dependencies between nested materialized views.
- GET_MV_DEPENDENCIES provides a list of the immediate (or direct) materialized view dependencies for an object.

Tips for Fast Refresh with UNION ALL

You can use fast refresh for materialized views that use the UNION ALL operator by providing a maintenance column in the definition of the materialized view. For example, a materialized view with a UNION ALL operator can be made fast refreshable as follows:

```
CREATE MATERIALIZED VIEW fast_rf_union_all_mv AS
SELECT x.rowid AS r1, y.rowid AS r2, a, b, c, 1 AS marker
FROM x, y WHERE x.a = y.b
UNION ALL
SELECT p.rowid, r.rowid, a, c, d, 2 AS marker
FROM p, r WHERE p.a = r.y;
```

The form of a maintenance marker column, column MARKER in the example, must be numeric_or_string_literal AS column_alias, where each UNION ALL member has a distinct value for numeric_or_string_literal.

Tips for Fast Refresh with Commit SCN-Based Materialized View Logs

You can often improve fast refresh performance by ensuring that your materialized view logs on the base table contain a WITH COMMIT SCN clause, often significantly. By optimizing materialized view log processing WITH COMMIT SCN, the fast refresh process can save time. The following example illustrates how to use this clause:

```
CREATE MATERIALIZED VIEW LOG ON sales WITH ROWID
 (prod_id, cust_id, time_id, channel_id, promo_id, quantity_sold, amount_sold),
COMMIT SCN INCLUDING NEW VALUES;
```

The materialized view refresh will then automatically use the commit SCN-based materialized view log to save refresh time.

Note that only new materialized view logs can take advantage of COMMIT SCN. Existing materialized view logs cannot be altered to add COMMIT SCN unless they are dropped and recreated.

When a materialized view is created on both base tables with timestamp-based materialized view logs and base tables with commit SCN-based materialized view logs, an error (ORA-32414) will be raised stating that materialized view logs are not compatible with each other for fast refresh.

Tips After Refreshing Materialized Views

After you have performed a load or incremental load and rebuilt the detail table indexes, you must re-enable integrity constraints (if any) and refresh the materialized views and materialized view indexes that are derived from that detail data. In a data warehouse environment, referential integrity constraints are normally enabled with the NOVALIDATE or RELY options. An important decision to make before performing a refresh operation is whether the refresh needs to be recoverable. Because materialized view data is redundant and can always be reconstructed from the detail tables, it might be preferable to disable logging on the materialized view. To disable logging and run incremental refresh non-recoverably, use the ALTER MATERIALIZED VIEW ... NOLOGGING statement prior to refreshing.

If the materialized view is being refreshed using the ON COMMIT method, then, following refresh operations, consult the alert log alert_*SID*.log and the trace file ora_*SID*_number.trc to check that no errors have occurred.

Using Materialized Views with Partitioned Tables

A major maintenance component of a data warehouse is synchronizing (refreshing) the materialized views when the detail data changes. Partitioning the underlying detail tables can reduce the amount of time taken to perform the refresh task. This is possible because partitioning enables refresh to use parallel DML to update the materialized view. Also, it enables the use of Partition Change Tracking.

Fast Refresh with Partition Change Tracking

In a data warehouse, changes to the detail tables can often entail partition maintenance operations, such as DROP, EXCHANGE, MERGE, and ADD PARTITION. To maintain the

materialized view after such operations used to require manual maintenance (see also CONSIDER FRESH) or complete refresh. You now have the option of using an addition to fast refresh known as Partition Change Tracking (PCT) refresh.

For PCT to be available, the detail tables must be partitioned. The partitioning of the materialized view itself has no bearing on this feature. If PCT refresh is possible, it will occur automatically and no user intervention is required in order for it to occur. See "Partition Change Tracking" on page 10-1 for PCT requirements.

The following examples illustrate the use of this feature. In "PCT Fast Refresh Scenario 1", assume sales is a partitioned table using the time_id column and products is partitioned by the prod_category column. The table times is not a partitioned table.

PCT Fast Refresh Scenario 1

1. The following materialized view satisfies requirements for PCT.

```
CREATE MATERIALIZED VIEW cust_mth_sales_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.time_id, s.prod_id, SUM(s.quantity_sold), SUM(s.amount_sold),
       p.prod_name, t.calendar_month_name, COUNT(*),
       COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY t.calendar_month_name, s.prod_id, p.prod_name, s.time_id;
```

2. You can use the DBMS_MVIEW.EXPLAIN_MVIEW procedure to determine which tables will allow PCT refresh.

```
MVNAME              CAPABILITY_NAME   POSSIBLE  RELATED_TEXT  MSGTXT
-----------------   ---------------   --------  ------------  ----------------
CUST_MTH_SALES_MV   PCT               Y         SALES
CUST_MTH_SALES_MV   PCT_TABLE         Y         SALES
CUST_MTH_SALES_MV   PCT_TABLE         N         PRODUCTS      no partition key
                                                              or PMARKER
                                                              in SELECT list
CUST_MTH_SALES_MV   PCT_TABLE         N         TIMES         relation is not
                                                              partitionedtable
```

   As can be seen from the partial sample output from EXPLAIN_MVIEW, any partition maintenance operation performed on the sales table will allow PCT fast refresh. However, PCT is not possible after partition maintenance operations or updates to the products table as there is insufficient information contained in cust_mth_sales_mv for PCT refresh to be possible. Note that the times table is not partitioned and hence can never allow for PCT refresh. Oracle Database will apply PCT refresh if it can determine that the materialized view has sufficient information to support PCT for all the updated tables. You can verify which partitions are fresh and stale with views such as DBA_MVIEWS and DBA_MVIEW_DETAIL_PARTITION.

   See "Analyzing Materialized View Capabilities" on page 9-29 for information on how to use this procedure and also some details regarding PCT-related views.

3. Suppose at some later point, a SPLIT operation of one partition in the sales table becomes necessary.

```
ALTER TABLE SALES
SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
```

```
INTO (PARTITION month3_1 TABLESPACE summ,
      PARTITION month3 TABLESPACE summ);
```

4. Insert some data into the `sales` table.
5. Fast refresh `cust_mth_sales_mv` using the `DBMS_MVIEW.REFRESH` procedure.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
   '',TRUE,FALSE,0,0,0,FALSE);
```

Fast refresh will automatically do a PCT refresh as it is the only fast refresh possible in this scenario. However, fast refresh will not occur if a partition maintenance operation occurs when any update has taken place to a table on which PCT is not enabled. This is shown in "PCT Fast Refresh Scenario 2".

"PCT Fast Refresh Scenario 1" would also be appropriate if the materialized view was created using the `PMARKER` clause as illustrated in the following:

```
CREATE MATERIALIZED VIEW cust_sales_marker_mv
BUILD IMMEDIATE
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) s_marker, SUM(s.quantity_sold),
  SUM(s.amount_sold), p.prod_name, t.calendar_month_name, COUNT(*),
  COUNT(s.quantity_sold), COUNT(s.amount_sold)
FROM sales s, products p, times t
WHERE  s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
         p.prod_name, t.calendar_month_name;
```

PCT Fast Refresh Scenario 2

In "PCT Fast Refresh Scenario 2", the first three steps are the same as in "PCT Fast Refresh Scenario 1" on page 16-24. Then, the `SPLIT` partition operation to the `sales` table is performed, but before the materialized view refresh occurs, records are inserted into the `times` table.

1. The same as in "PCT Fast Refresh Scenario 1".
2. The same as in "PCT Fast Refresh Scenario 1".
3. The same as in "PCT Fast Refresh Scenario 1".
4. After issuing the same `SPLIT` operation, as shown in "PCT Fast Refresh Scenario 1", some data will be inserted into the `times` table.

```
ALTER TABLE SALES
SPLIT PARTITION month3 AT (TO_DATE('05-02-1998', 'DD-MM-YYYY'))
INTO (PARTIITION month3_1 TABLESPACE summ,
      PARTITION month3 TABLESPACE summ);
```

5. Refresh `cust_mth_sales_mv`.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F',
    '',TRUE,FALSE,0,0,0,FALSE);
ORA-12052: cannot fast refresh materialized view SH.CUST_MTH_SALES_MV
```

The materialized view is not fast refreshable because DML has occurred to a table on which PCT fast refresh is not possible. To avoid this occurring, Oracle recommends performing a fast refresh immediately after any partition maintenance operation on detail tables for which partition tracking fast refresh is available.

If the situation in "PCT Fast Refresh Scenario 2" occurs, there are two possibilities; perform a complete refresh or switch to the CONSIDER FRESH option outlined in the following, if suitable. However, it should be noted that CONSIDER FRESH and partition change tracking fast refresh are not compatible. Once the ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH statement has been issued, PCT refresh will no longer be applied to this materialized view, until a complete refresh is done. Moreover, you should not use CONSIDER FRESH unless you have taken manual action to ensure that the materialized view is indeed fresh.

A common situation in a data warehouse is the use of rolling windows of data. In this case, the detail table and the materialized view may contain say the last 12 months of data. Every month, new data for a month is added to the table and the oldest month is deleted (or maybe archived). PCT refresh provides a very efficient mechanism to maintain the materialized view in this case.

PCT Fast Refresh Scenario 3

1. The new data is usually added to the detail table by adding a new partition and exchanging it with a table containing the new data.

```
ALTER TABLE sales ADD PARTITION month_new ...
ALTER TABLE sales EXCHANGE PARTITION month_new month_new_table
```

2. Next, the oldest partition is dropped or truncated.

```
ALTER TABLE sales DROP PARTITION month_oldest;
```

3. Now, if the materialized view satisfies all conditions for PCT refresh.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F', '', TRUE,
FALSE,0,0,0,FALSE);
```

Fast refresh will automatically detect that PCT is available and perform a PCT refresh.

Fast Refresh with CONSIDER FRESH

In a data warehouse, you may often wish to accumulate historical information in the materialized view even though this information is no longer in the detailed tables. In this case, you could maintain the materialized view using the ALTER MATERIALIZED VIEW *materialized_view_name* CONSIDER FRESH statement.

Note that CONSIDER FRESH declares that the contents of the materialized view are FRESH (in sync with the detail tables). Care must be taken when using this option in this scenario in conjunction with query rewrite because you may see unexpected results.

After using CONSIDER FRESH in an historical scenario, you will be able to apply traditional fast refresh after DML and direct loads to the materialized view, but not PCT fast refresh. This is because if the detail table partition at one time contained data that is currently kept in aggregated form in the materialized view, PCT refresh in attempting to resynchronize the materialized view with that partition could delete historical data which cannot be recomputed.

Assume the sales table stores the prior year's data and the cust_mth_sales_mv keeps the prior 10 years of data in aggregated form.

1. Remove old data from a partition in the sales table:

```
ALTER TABLE sales TRUNCATE PARTITION month1;
```

The materialized view is now considered stale and requires a refresh because of the partition operation. However, as the detail table no longer contains all the data associated with the partition fast refresh cannot be attempted.

2. Therefore, alter the materialized view to tell Oracle Database to consider it fresh.

```
ALTER MATERIALIZED VIEW cust_mth_sales_mv CONSIDER FRESH;
```

This statement informs Oracle Database that `cust_mth_sales_mv` is fresh for your purposes. However, the materialized view now has a status that is neither known fresh nor known stale. Instead, it is `UNKNOWN`. If the materialized view has query rewrite enabled in `QUERY_REWRITE_INTEGRITY = stale_tolerated` mode, it will be used for rewrite.

3. Insert data into `sales`.

4. Refresh the materialized view.

```
EXECUTE DBMS_MVIEW.REFRESH('CUST_MTH_SALES_MV', 'F', '', TRUE,
FALSE,0,0,0,FALSE);
```

Because the fast refresh detects that only `INSERT` statements occurred against the sales table it will update the materialized view with the new data. However, the status of the materialized view will remain `UNKNOWN`. The only way to return the materialized view to `FRESH` status is with a complete refresh which, also will remove the historical data from the materialized view.

17

Change Data Capture

Change Data Capture efficiently identifies and captures data that has been added to, updated in, or removed from, Oracle relational tables and makes this change data available for use by applications or individuals.

> **Note:** Oracle Change Data Capture will be de-supported in a future release of Oracle Database and will be replaced with Oracle GoldenGate. Therefore, Oracle strongly recommends that you use Oracle GoldenGate for new applications.
>
> For Oracle Database 11*g* Release 2 (11.2), Change Data Capture continues to function as in earlier releases. If you are currently using Change Data Capture, then you will be able to continue to do so for the foreseeable future. However, Change Data Capture will not be further enhanced, and will only be supported based on the current, documented functionality.

This chapter describes Change Data Capture in the following sections:

See *Oracle Database PL/SQL Packages and Types Reference* for reference information about the Change Data Capture publish and subscribe PL/SQL packages.

Overview of Change Data Capture

Often, data warehousing involves the extraction and transportation of relational data from one or more production databases into a data warehouse for analysis. Change Data Capture quickly identifies and processes only the data that has changed and makes the change data available for further use.

Capturing Change Data Without Change Data Capture

Prior to the introduction of Change Data Capture, there were a number of ways that users could capture change data, including table differencing and change-value selection.

Table differencing involves transporting a copy of an entire table from the source (production) database to the staging database (where the change data is captured), where an older version of the table already exists. Using the SQL `MINUS` operator, you can obtain the inserted and new versions of updated rows with the following query:

```
SELECT * FROM new_version
MINUS SELECT * FROM old_version;
```

Moreover, you can obtain the deleted rows and old versions of updated rows with the following query:

```
SELECT * FROM old_version
MINUS SELECT * FROM new_version;
```

However, there are several problems with this method:

- It requires that the new version of the entire table be transported to the staging database, not just the change data, thereby greatly increasing transport costs.
- The computational cost of performing the two `MINUS` operations on the staging database can be very high.
- Table differencing cannot capture data that have reverted to their old values. For example, suppose the price of a product changes several times between the old version and the new version of the product's table. If the price in the new version ends up being the same as the old, table differencing cannot detect that the price has fluctuated. Moreover, any intermediate price values between the old and new versions of the product's table cannot be captured using table differencing.
- There is no way to determine which changes were made as part of the same transaction. For example, suppose a sales manager creates a special discount to close a deal. The fact that the creation of the discount and the creation of the sale occurred as part of the same transaction cannot be captured, unless the source database is specifically designed to do so.

Change-value selection involves capturing the data on the source database by selecting the new and changed data from the source tables based on the value of a specific column. For example, suppose the source table has a `LAST_UPDATE_DATE` column. To capture changes, you base your selection from the source table on the `LAST_UPDATE_DATE` column value.

However, there are also several limitations with this method:

- The overhead of capturing the change data must be borne on the source database, and you must run potentially expensive queries against the source table on the source database. The need for these queries may force you to add indexes that would otherwise be unneeded. There is no way to offload this overhead to the staging database.

- This method is no better at capturing intermediate values than the table differencing method. If the price in the product's table fluctuates, you will not be able to capture all the intermediate values, or even tell if the price had changed, if the ending value is the same as it was the last time that you captured change data.
- This method is also no better than the table differencing method at capturing which data changes were made together in the same transaction. If you need to capture information concerning which changes occurred together in the same transaction, you must include specific designs for this purpose in your source database.
- The granularity of the change-value column may not be fine enough to uniquely identify the new and changed rows. For example, suppose the following:
  - You capture data changes using change-value selection on a date column such as `LAST_UPDATE_DATE`.
  - The capture happens at a particular instant in time, 14-FEB-2003 17:10:00.
  - Additional updates occur to the table during the same second that you performed your capture.

  When you next capture data changes, you will select rows with a `LAST_UPDATE_DATE` strictly after 14-FEB-2003 17:10:00, and thereby miss the changes that occurred during the remainder of that second.

  To use change-value selection, you either have to accept that anomaly, add an artificial change-value column with the granularity you need, or lock out changes to the source table during the capture process, thereby further burdening the performance of the source database.
- You have to design your source database in advance with this capture mechanism in mind – all tables from which you wish to capture change data must have a change-value column. If you want to build a data warehouse with data sources from legacy systems, those legacy systems may not supply the necessary change-value columns you need.

Change Data Capture does not depend on expensive and cumbersome table differencing or change-value selection mechanisms. Instead, it captures the change data resulting from `INSERT`, `UPDATE`, and `DELETE` operations made to user tables. The change data is then stored in a relational table called a change table, and the change data is made available to applications or individuals in a controlled way.

Capturing Change Data with Change Data Capture

Change Data Capture can capture and publish committed change data in either of the following modes:

- **Synchronous**

  Triggers on the source database allow change data to be captured immediately, as each SQL statement that performs a data manipulation language (DML) operation (`INSERT`, `UPDATE`, or `DELETE`) is made. In this mode, change data is captured as part of the transaction modifying the source table. Synchronous Change Data Capture is available with Oracle Standard Edition and Enterprise Edition. This mode is described in detail in "Synchronous Change Data Capture" on page 17-8.

- **Asynchronous**

  By taking advantage of the data sent to the redo log files, change data is captured after a SQL statement that performs a DML operation is committed. In this mode,

change data is not captured as part of the transaction that is modifying the source table, and therefore has no effect on that transaction.

There are three modes of asynchronous Change Data Capture: HotLog, Distributed HotLog, and AutoLog. These modes are described in detail in "Asynchronous Change Data Capture" on page 17-9.

Asynchronous Change Data Capture is built on, and provides a relational interface to, Oracle Streams. See *Oracle Streams Concepts and Administration* for information on Oracle Streams.

The following list describes the advantages of capturing change data with Change Data Capture:

- Completeness

  Change Data Capture can capture all effects of `INSERT`, `UPDATE`, and `DELETE` operations, including data values before and after `UPDATE` operations.

- Performance

  Asynchronous Change Data Capture can be configured to have minimal performance impact on the source database.

- Interface

  Change Data Capture includes the `DBMS_CDC_PUBLISH` and `DBMS_CDC_SUBSCRIBE` packages, which provide easy-to-use publish and subscribe interfaces.

- Cost

  Change Data Capture reduces overhead cost because it simplifies the extraction of change data from the database and is part of the Oracle Database.

Note that you cannot use any table that uses transparent data encryption as a source table for synchronous Change Data Capture. Asynchronous Change Data Capture supports transparent data encryption if both the source and staging databases have `COMPATIBLE` set to 11 or higher. Change Data Capture does not encrypt the data in the change table. A user who wants to encrypt a column in the change table can manually use an `ALTER TABLE` statement to encrypt the column in the change table. See *Oracle Streams Concepts and Administration* for information on Oracle Streams.

A Change Data Capture system is based on the interaction of publishers and subscribers to capture and distribute change data, as described in the next section.

Publish and Subscribe Model

Most Change Data Capture systems have one person who captures and publishes change data; this person is the **publisher**. There can be multiple applications or individuals that access the change data; these applications and individuals are the **subscribers**. Change Data Capture provides PL/SQL packages to accomplish the publish and subscribe tasks.

The following sections describe the roles of the publisher and subscriber in detail. Subsequent sections describe change sources, more about modes of Change Data Capture, and change tables.

Publisher

The publisher is usually a database administrator (DBA) who creates and maintains the schema objects that make up the Change Data Capture system. For all modes of Change Data Capture, except Distributed HotLog, there is typically one publisher on

the staging database. For the Distributed HotLog mode of Change Data Capture there needs to be a publisher defined on the source and staging databases. The following list describes the source and staging databases and the objects of interest to Change Data Capture on each:

- **Source database**

  This is the production database that contains the data of interest. The following objects of interest to Change Data Capture reside on the source database:

  – The source tables

    The **source tables** are the production database tables that contain the data of interest. They may be all or a subset of the source database tables.

  – Redo log files

    For asynchronous modes of change data capture, the change data is collected from either the online or archived redo log files (or both). For asynchronous AutoLog mode, archived redo log files are copied from the source database to the staging database.

  – Change source

    The **change source** is a logical representation of the source database. The method for representing the source varies, depending on the mode of Change Data Capture.

    For the asynchronous Distributed HotLog mode of Change Database capture only, the change source resides on the source database. For the other modes of Change Data Capture, the change source resides on the staging database.

    See "Asynchronous Distributed HotLog Mode" on page 17-10 for details about the Distributed HotLog change source.

- **Staging database**

  This is the database to which the captured change data is applied. Depending on the capture mode that the publisher uses, the staging database can be the same as, or different from, the source database. The following Change Data Capture objects reside on the staging database:

  – Change table

    A **change table** is a relational table into which change data for a single source table is loaded. To subscribers, a change table is known as a **publication**.

  – Change set

    A **change set** is a set of change data that is guaranteed to be transactionally consistent. It contains one or more change tables.

  – Change source

    The change source for the following modes of Change Data Capture reside on the staging database:

    * Synchronous - See "Synchronous Change Data Capture" on page 17-8 for details.

    * Asynchronous HotLog - See "Asynchronous HotLog Mode" on page 17-10 for details.

    * Asynchronous AutoLog - See "Asynchronous AutoLog Mode" on page 17-11 for details.

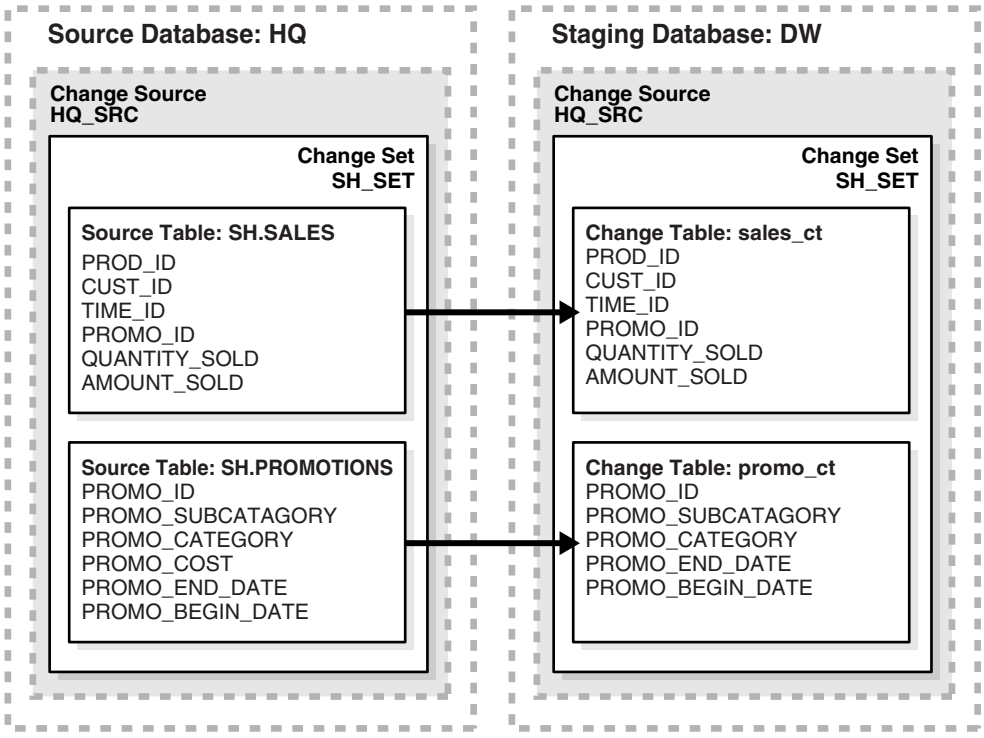
These are the main tasks performed by the publisher:

- Determines the source databases and tables from which the subscribers are interested in viewing change data, and the mode (synchronous or one of the asynchronous modes) in which to capture the change data.
- Uses the Oracle-supplied package, DBMS_CDC_PUBLISH, to set up the system to capture change data from the source tables of interest.
- Allows subscribers to have controlled access to the change data in the change tables by using the SQL GRANT and REVOKE statements to grant and revoke the SELECT privilege on change tables for users and roles. (Keep in mind, however, that subscribers use views, not change tables directly, to access change data.)

In Figure 17–1, the publisher determines that subscribers are interested in viewing change data from the HQ source database. In particular, subscribers are interested in change data from the sh.sales and sh.promotions source tables.

The publisher decides to use the asynchronous AutoLog mode of capturing change data. On the DW staging database, he creates a change source HQ_SRC, a change set, SH_SET, and two change tables: sales_ct and promo_ct. The sales_ct change table contains all the columns from the source table, sh.sales. For the promo_ct change table, however, the publisher has decided to exclude the PROMO_COST column.

Figure 17–1 Publisher Components in a Change Data Capture System



Subscribers

The subscribers are consumers of the published change data. A subscriber performs the following tasks:

- Uses the Oracle supplied package, DBMS_CDC_SUBSCRIBE, to:
  - Create subscriptions

A **subscription** controls access to the change data from one or more source tables of interest within a single change set. A subscription contains one or more subscriber views.

A **subscriber view** is a view that specifies the change data from a specific publication in a subscription. The subscriber is restricted to seeing change data that the publisher has published and has granted the subscriber access to use. See "Subscribing to Change Data" on page 17-44 for more information on choosing a method for specifying a subscriber view.

  - Notify Change Data Capture when ready to receive a set of change data

    A **subscription window** defines the time range of rows in a publication that the subscriber can currently see in subscriber views. The oldest row in the window is called the **low boundary**; the newest row in the window is called the **high boundary**. Each subscription has its own subscription window that applies to all of its subscriber views.

  - Notify Change Data Capture when finished with a set of change data

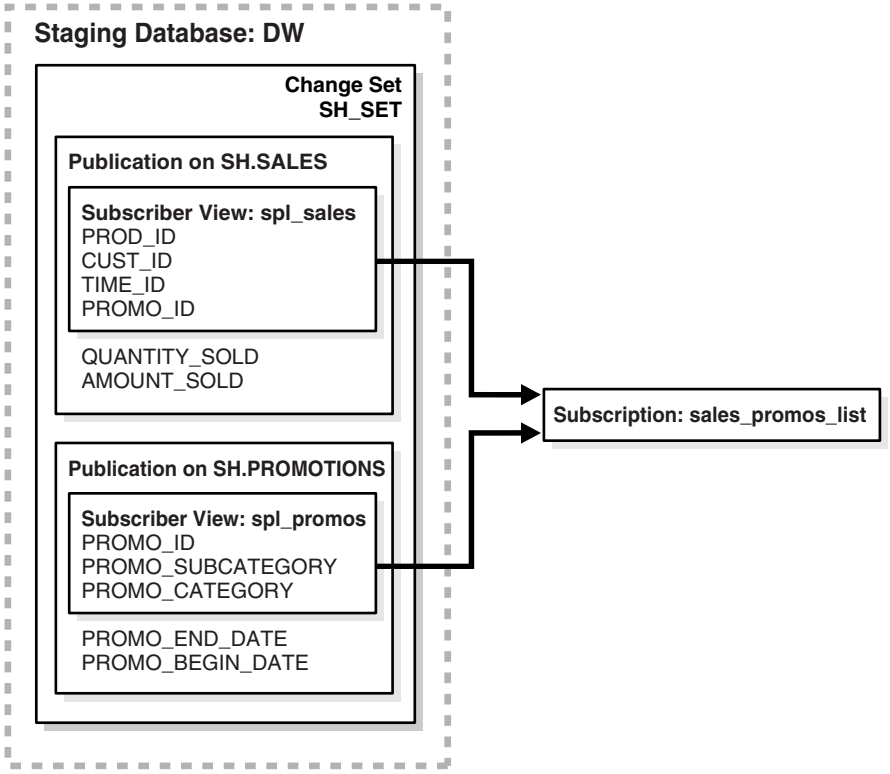
- Uses `SELECT` statements to retrieve change data from the subscriber views.

A subscriber has the privileges of the user account under which the subscriber is running, plus any additional privileges that have been granted to the subscriber.

In Figure 17–2, the subscriber is interested in a subset of columns that the publisher (in Figure 17–1) has published. Note that the *publications* shown in Figure 17–2, are represented as *change tables* in Figure 17–1; this reflects the different terminology used by subscribers and publishers, respectively.

The subscriber creates a subscription, `sales_promos_list` and two subscriber views (`spl_sales` and `spl_promos`) on the `SH_SET` change set on the DW staging database. Within each subscriber view, the subscriber includes a subset of the columns that were made available by the publisher. Note that because the publisher did not create a change table that includes the `PROMO_COST` column, there is no way for the subscriber to view change data for that column. The subscriber need not be aware of the mode of change data capture employed by the publisher.

Figure 17–2 Subscriber Components in a Change Data Capture System



Change Data Capture provides the following benefits for subscribers:

- Guarantees that each subscriber sees all the changes
- Keeps track of multiple subscribers and gives each subscriber shared access to change data
- Handles all the storage management by automatically removing data from change tables when it is no longer required by any of the subscribers. Keep in mind that Change Data Capture starts a job in the job queue that runs once every 24 hours for handling purging. Many things can go wrong with this job (such as if it is deleted or the schedule is changed), so this automatic processing depends on the job queue process being up and running and the Change Data Capture job being there. Also, in logical standby environments, the purge job is not submitted.

> **Note:** Oracle provides the previously listed benefits only when the subscriber accesses change data through a subscriber view.

Change Sources and Modes of Change Data Capture

Change Data Capture provides synchronous and asynchronous modes for capturing change data. The following sections summarize how each mode of Change Data Capture is performed, and the change source associated with each mode of Change Data Capture.

Synchronous Change Data Capture

The synchronous mode uses triggers on the source database to capture change data. It has no latency because the change data is captured continuously and in real time on

the source database. The change tables are populated when DML operations on the source table are committed.

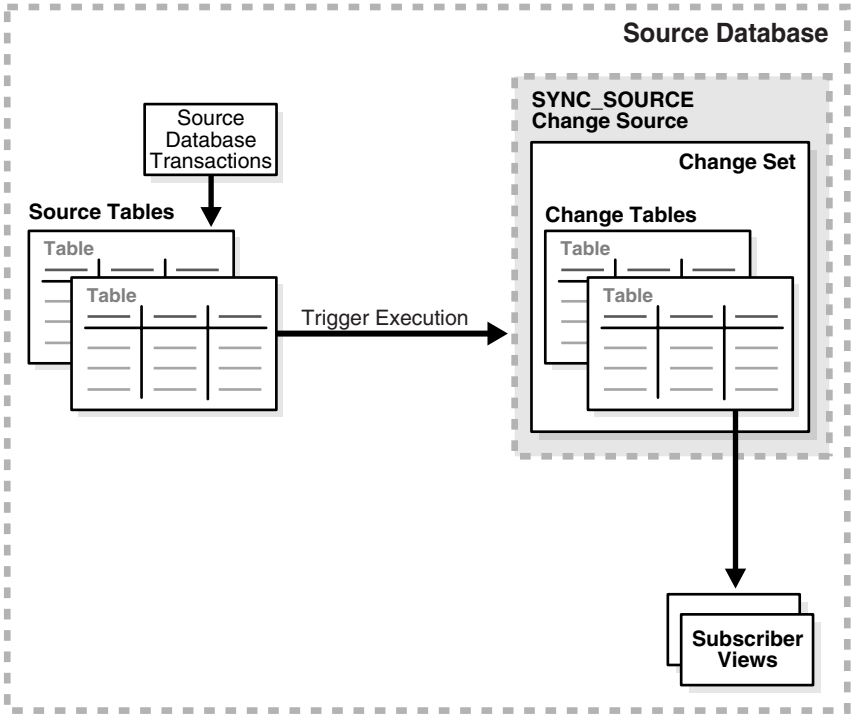
There is a single, predefined synchronous change source, SYNC_SOURCE, that represents the source database. This is the only synchronous change source. It cannot be altered or dropped.

While the synchronous mode of Change Data Capture adds overhead to the source database at capture time, this mode can reduce costs (as compared to attempting to extract change data using table differencing or change-value section) by simplifying the extraction of change data.

Change tables for this mode of Change Data Capture must reside locally in the source database.

Figure 17–3 illustrates the synchronous configuration. Triggers executed after DML operations occur on the source tables populate the change tables in the change sets within the SYNC_SOURCE change source.

Figure 17–3 Synchronous Change Data Capture Configuration



Asynchronous Change Data Capture

The asynchronous modes capture change data from the database redo log files after changes have been committed to the source database.

The asynchronous modes of Change Data Capture are dependent on the level of supplemental logging enabled at the source database. Supplemental logging adds redo logging overhead at the source database, so it must be carefully balanced with the needs of the applications or individuals using Change Data Capture. See "Asynchronous Change Data Capture and Supplemental Logging" on page 17-70 for information on supplemental logging.

The three modes of capturing change data are described in the following sections:

- Asynchronous HotLog Mode
- Asynchronous Distributed HotLog Mode
- Asynchronous AutoLog Mode

Asynchronous HotLog Mode

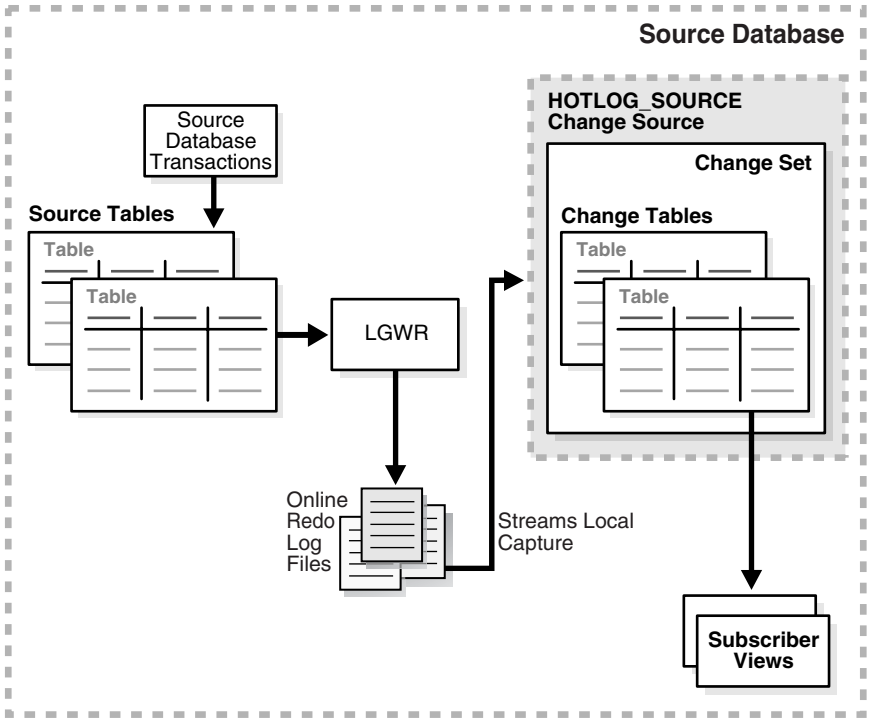
In the asynchronous HotLog mode, change data is captured from the online redo log file on the source database. There is a brief latency between the act of committing source table transactions and the arrival of change data.

There is a single, predefined HotLog change source, `HOTLOG_SOURCE`, that represents the current online redo log files of the source database. This is the only HotLog change source. It cannot be altered or dropped.

Change tables for this mode of Change Data Capture must reside locally in the source database.

Figure 17–4 illustrates the asynchronous HotLog configuration. The Logwriter Process (`LGWR`) records committed transactions in the online redo log files on the source database. Change Data Capture uses Oracle Streams processes to automatically populate the change tables in the change sets within the `HOTLOG_SOURCE` change source as newly committed transactions arrive.

Figure 17–4 Asynchronous HotLog Configuration



Asynchronous Distributed HotLog Mode

In the asynchronous Distributed HotLog mode, change data is captured from the online redo log file on the source database.

There is no predefined Distributed HotLog change source. Unlike other modes of Change Data Capture, the Distributed HotLog mode splits change data capture

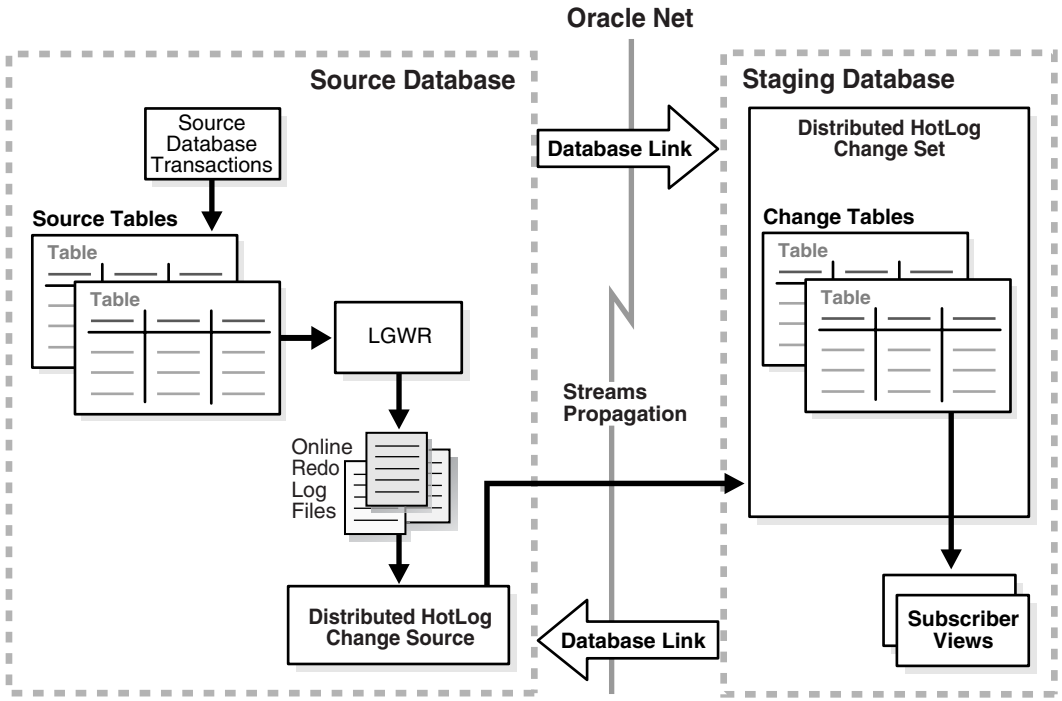
activities and objects across the source and staging database. Change sources are defined on the source database by the staging database publisher.

A Distributed HotLog change source represents the current online redo log files of the source database. However, staging database publishers can define multiple Distributed HotLog change sources, each of which contains change sets on a different staging database. The source and staging database can be on different hardware platforms and be running different operating systems, however some restrictions apply. See "Summary of Supported Distributed HotLog Configurations and Restrictions" on page 17-73 for information on these restrictions.

Figure 17–5 illustrates the asynchronous Distributed HotLog configuration. The change source on the source database captures change data from the online redo log files and uses Streams to propagate it to the change set on the staging database. The change set on the staging database populates the change tables within the change set.

There are two publishers required for this mode of Change Data Capture, one on the source database and one on the staging database. The source database publisher defines a database link on the source database to connect to the staging database as the staging database publisher. The staging database publisher defines a database link on the staging database to connect to the source database on the source database publisher. All publishing operations are performed by the staging database publisher. See "Performing Asynchronous Distributed HotLog Publishing" on page 17-32 for details.

Figure 17–5 Asynchronous Distributed HotLog Configuration



Asynchronous AutoLog Mode

In the asynchronous AutoLog mode, change data is captured from a set of redo log files managed by redo transport services. **Redo transport services** control the automated transfer of redo log files from the source database to the staging database. Using database initialization parameters (described in "Initialization Parameters for Asynchronous AutoLog Publishing" on page 17-23), the publisher configures redo

transport services to copy the redo log files from the source database system to the staging database system and to automatically register the redo log files. Asynchronous AutoLog mode can obtain change data from either the source database online redo log or from source database archived redo logs. These options are known as asynchronous AutoLog online and asynchronous AutoLog archive.

With the AutoLog online option, redo transport services is set up to copy redo data from the online redo log at the source database to the standby redo log at the staging database. Change sets are populated after individual source database transactions commit. There can only be one AutoLog online change source on a given staging database and it can contain only one change set.

With the AutoLog archive option, redo transport services is set up to copy archived redo logs from the source database to the staging database. Change sets are populated as new archived redo log files arrive on the staging database. The degree of latency depends on the frequency of redo log file switches on the source database. The AutoLog archive option has a higher degree of latency than the AutoLog online option, but there can be as many AutoLog archive change sources as desired on a given staging database.

There is no predefined AutoLog change source. The publisher provides information about the source database to create an AutoLog change source. See "Performing Asynchronous AutoLog Publishing" on page 17-38 for details.

Figure 17–6 shows a Change Data Capture asynchronous AutoLog online configuration in which the LGWR process on the source database copies redo data to both the online redo log file on the source database and to the standby redo log files on the staging database as specified by the `LOG_ARCHIVE_DEST_2` parameter. (Although the image presents this parameter as `LOG_ARCHIVE_DEST_2`, the integer value can be any value between 1 and 10.)

Note that the LGWR process uses Oracle Net to send redo data over the network to the remote file server (RFS) process. Transmitting redo data to a remote destination requires uninterrupted connectivity through Oracle Net.

On the staging database, the RFS process writes the redo data to the standby redo log files. Then, Change Data Capture uses Oracle Streams downstream capture to populate the change tables in the change sets within the AutoLog change source.

The source database and the staging database must be running on the same hardware, operating system, and Oracle version.

Figure 17–6 Asynchronous Autolog Online Change Data Capture Configuration

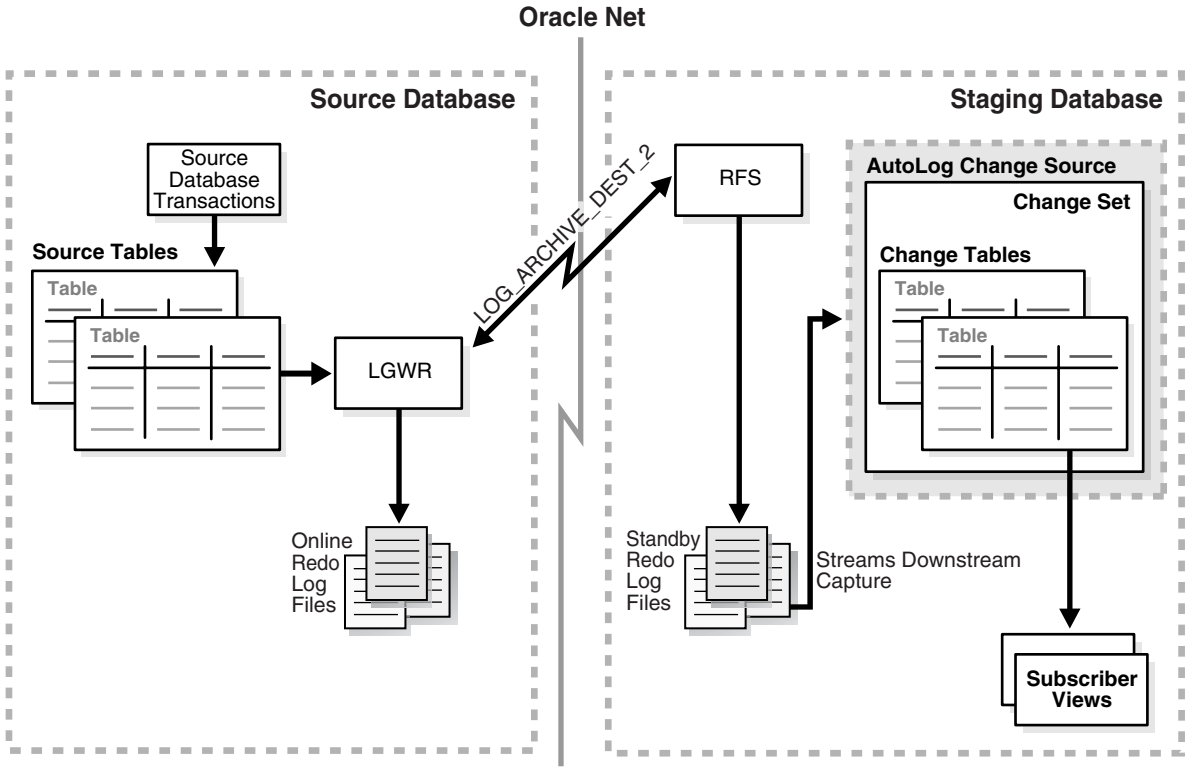


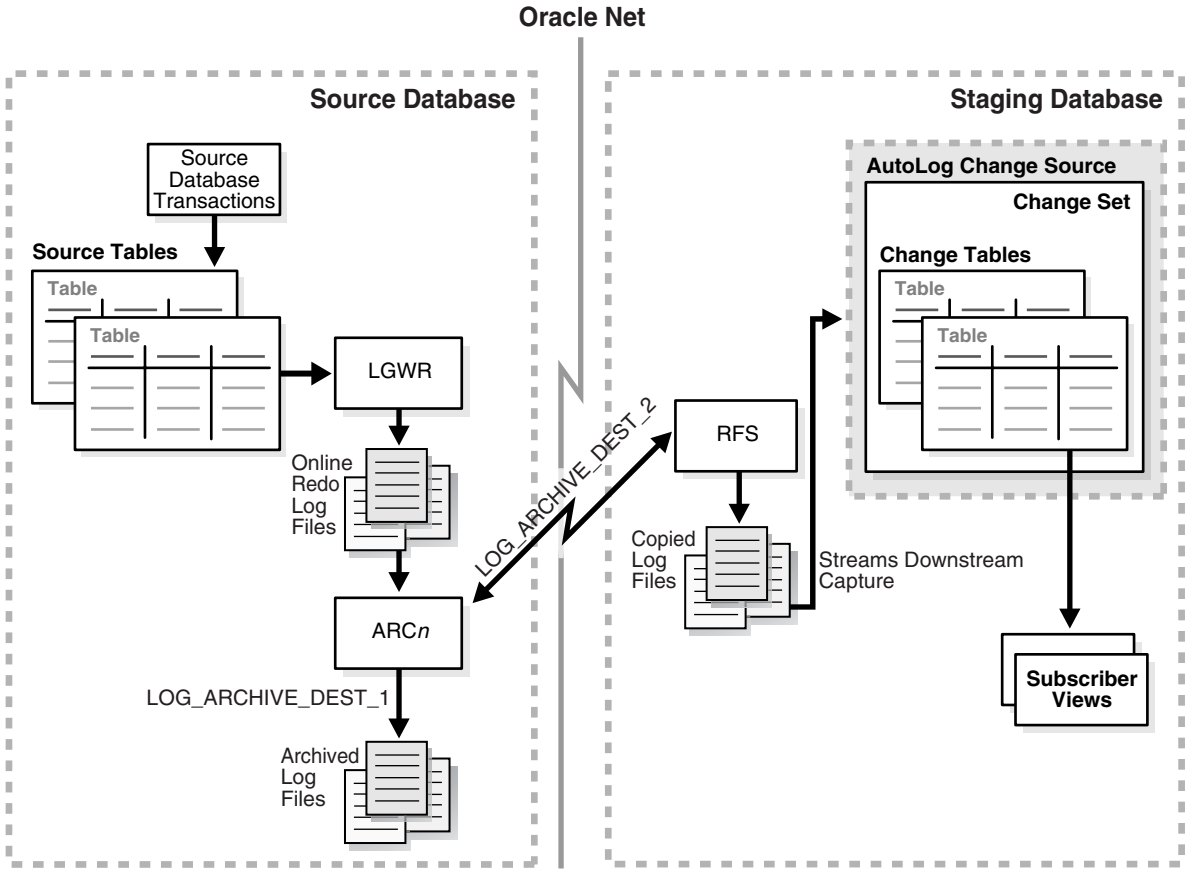
Figure 17–7 shows a typical Change Data Capture asynchronous AutoLog archive configuration in which, when the redo log file switches on the source database, archiver processes archive the redo log file on the source database to the destination specified by the `LOG_ARCHIVE_DEST_1` parameter and copy the redo log file to the staging database as specified by the `LOG_ARCHIVE_DEST_2` parameter. (Although the image presents these parameters as `LOG_ARCHIVE_DEST_1` and `LOG_ARCHIVE_DEST_2`, the integer value in these parameter strings can be any value between 1 and 10.)

Note that the archiver processes use Oracle Net to send redo data over the network to the remote file server (RFS) process. Transmitting redo log files to a remote destination requires uninterrupted connectivity through Oracle Net.

On the staging database, the RFS process writes the redo data to the copied log files. Then, Change Data Capture uses Oracle Streams downstream capture to populate the change tables in the change sets within the AutoLog change source.

See *Oracle Data Guard Concepts and Administration* for more information regarding Redo Transport Services.

Figure 17–7 Asynchronous AutoLog Archive Change Data Capture Configuration



Change Sets

A **change set** is a logical grouping of change data that is guaranteed to be transactionally consistent and that can be managed as a unit. A change set is a member of one (and only one) change source.

Note: Change Data Capture change sources can contain one or more change sets with the following restrictions:

1. All of the change sets for a Distributed HotLog change source must be on the same staging database
2. An AutoLog online change source can only contain one change set

---

When a publisher includes two or more change tables in the same change set, subscribers can perform join operations across the tables represented within the change set and be assured of transactional consistency.

Conceptually, a change set shares the same mode as its change source. For example, an AutoLog change set is a change set contained in an AutoLog change source. Publishers define change sets using the `DBMS_CDC_PUBLISH.CREATE_CHANGE_SET` package. In the case of synchronous Change Data Capture, the publisher can also use a predefined change set, `SYNC_SET`. The `SYNC_SET` change set, however, cannot be altered or dropped.

To keep the change tables in the change set from growing larger indefinitely, publishers can purge unneeded change data from change tables at the change set level. See "Purging Change Tables of Unneeded Data" on page 17-61 for more information on purging change data.

Valid Combinations of Change Sources and Change Sets

Table 17–1 summarizes the valid combinations of change sources and change sets and indicates whether each is predefined or publisher-defined. In addition, it indicates whether the source database represented by the change source is local to or remote from the staging database, and whether the change source is used for synchronous or asynchronous Change Data Capture.

Table 17–1 Summary of Change Sources and Change Sets

Mode	Change Source	Source Database Represented	Associated Change Sets
Synchronous	Predefined `SYNC_SOURCE`	Local	Predefined `SYNC_SET` and publisher-defined
Asynchronous HotLog	Predefined `HOTLOG_SOURCE`	Local	Publisher-defined
Asynchronous Distributed HotLog	Publisher-defined	Remote	Publisher-defined. Change sets must all be on the same staging database
Asynchronous AutoLog online	Publisher-defined	Remote	Publisher-defined. There can only be one change set in an AutoLog online change source
Asynchronous AutoLog archive	Publisher-defined	Remote	Publisher-defined

Change Tables

A given change table contains the change data resulting from DML operations performed on a given source table. A change table consists of two things: the change data itself, which is stored in a database table; and the system metadata necessary to maintain the change table, which includes control columns.

The publisher specifies the source columns that are to be included in the change table. Typically, for a change table to contain useful data, the publisher needs to include the primary key column in the change table along with any other columns of interest to subscribers. For example, suppose subscribers are interested in changes that occur to the `UNIT_COST` and the `UNIT_PRICE` columns in the `sh.costs` table. If the publisher does not include the `PROD_ID` column in the change table, subscribers will know only that the unit cost and unit price of some products have changed, but will be unable to determine for which products these changes have occurred.

There are optional and required control columns. The required control columns are always included in a change table; the optional ones are included if specified by the publisher when creating the change table. Control columns are managed by Change Data Capture. See "Understanding Change Table Control Columns" on page 17-55 and "Understanding TARGET_COLMAP$ and SOURCE_COLMAP$ Values" on page 17-57 for detailed information on control columns.

Getting Information About the Change Data Capture Environment

Information about the Change Data Capture environment is provided in the static data dictionary views described in Table 17–2 and Table 17–3. Table 17–2 lists the views that are intended for use by publishers; the user must have the SELECT_CATALOG_ROLE privilege to access the views listed in this table. Table 17–3 lists the views that are intended for use by subscribers. Table 17–3 includes views with the prefixes ALL and USER. These prefixes have the following general meanings:

- A view with the ALL prefix allows the user to display all the information accessible to the user, including information from the current user's schema as well as information from objects in other schemas, if the current user has access to those objects by way of grants of privileges or roles.
- A view with the USER prefix allows the user to display all the information from the schema of the user issuing the query without the use of additional special privileges or roles.

Note: To look at all the views (those intended for both the publisher and the subscriber), a user must have the SELECT_CATALOG_ROLE privilege.

Table 17–2 Views Intended for Use by Change Data Capture Publishers

View Name	Description
ALL_CHANGE_SOURCES	Describes existing change sources.
ALL_CHANGE_PROPAGATIONS	Describes the Oracle Streams propagation associated with a given Distributed HotLog change source on the source database. This view is populated on the source database for 11.1 or 11.2 change sources or on the staging database for 9.2, 10.1 or 10.2 change sources.
ALL_CHANGE_PROPAGATION_SETS	Describes the Oracle Streams propagation associated with a given Distributed HotLog change set on the staging database. This view is populated on the source database for 11.1 or 11.2 change sources or on the staging database for 9.2, 10.1 or 10.2 change sources.
ALL_CHANGE_SETS	Describes existing change sets.
ALL_CHANGE_TABLES	Describes existing change tables.
DBA_SOURCE_TABLES	Describes all published source tables in the database.
DBA_PUBLISHED_COLUMNS	Describes all published columns of source tables in the database.
DBA_SUBSCRIPTIONS	Describes all subscriptions.
DBA_SUBSCRIBED_TABLES	Describes all source tables to which any subscriber has subscribed.
DBA_SUBSCRIBED_COLUMNS	Describes the columns of source tables to which any subscriber has subscribed.

Table 17–3 Views Intended for Use by Change Data Capture Subscribers

View Name	Description
ALL_SOURCE_TABLES	Describes all public source tables for change tables that are owned by the current user.

Table 17–3 (Cont.) Views Intended for Use by Change Data Capture Subscribers

View Name	Description
USER_SOURCE_TABLES	Describes all public source tables for change tables that are owned by the current user.
ALL_PUBLISHED_COLUMNS	Describes all published columns of source tables for change tables that are owned by the current user.[1]
USER_PUBLISHED_COLUMNS	Describes all published columns of source tables for change tables that are owned by the current user.
ALL_SUBSCRIPTIONS	Describes all of the subscriptions created by the current user.
USER_SUBSCRIPTIONS	Describes all of the subscriptions created by the current user.
ALL_SUBSCRIBED_TABLES	Describes the source tables to which the current user has subscribed.
USER_SUBSCRIBED_TABLES	Describes the source tables to which the current user has subscribed.
ALL_SUBSCRIBED_COLUMNS	Describes the columns of source tables to which the current user has subscribed.
USER_SUBSCRIBED_COLUMNS	Describes the columns of source tables to which the current user has subscribed.

[1] ALL_PUBLISHED_COLUMNS, USER_PUBLISHED_COLUMNS, and the view for *_SOURCE_TABLES are synonyms, so the user must have ownership to view the table or column information.

See *Oracle Database Reference* for complete information about these views.

Preparing to Publish Change Data

This section describes the tasks the publisher should perform before starting to publish change data, information on creating publishers, information on selecting a mode in which to capture change data, instructions on setting up database links required for the asynchronous Distributed HotLog mode of Change Data Capture, and instructions on setting database initialization parameters required by Change Data Capture.

A publisher should do the following before performing the actual steps for publishing:

- Gather requirements from the subscribers.
- Determine which source database contains the relevant source tables.
- Choose the capture mode: synchronous, asynchronous HotLog, asynchronous Distributed HotLog, or asynchronous AutoLog, as described in "Determining the Mode in Which to Capture Data" on page 17-19.
- Ensure that the source and staging database DBAs have set database initialization parameters, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20 and "Publishing Change Data" on page 17-26.
- Sets up database links from the source database to the staging database and from the staging database to the source database, as shown in "Performing Asynchronous Distributed HotLog Publishing" on page 17-32. Be aware that this requires the source database publisher to know the username and password of the staging database publisher and the staging database publisher to know the username and password of the source database publisher.

Creating a User to Serve As a Publisher

For all modes of Change Database Capture, the staging database DBA creates a user to serve as a publisher for Change Data Capture. In addition, for the asynchronous Distributed HotLog mode of Change Data Capture, the source database DBA also creates a user to serve as a publisher. On the source database, this publisher's only task is to create a database link from the source database to the staging database.

The SYS and SYSTEM users cannot be used as a Change Data Capture publisher, and a Change Data Capture publisher should not use the SYSTEM tablespace as its default tablespace.

The following sections describe how to set up a publisher as required for each mode of Change Data Capture.

Note: If a publisher is dropped with a SQL DROP USER CASCADE statement, then all Change Data Capture objects owned by that publisher are dropped, except those that contain Change Data Capture objects owned by other publishers.

For example, suppose publisher CDCPUB1 owns the change set CDCPUB1_SET that contains the change table CDCPUB2.SALES_CT. Issuing a DROP USER CASCADE statement to drop CDCPUB1 does not result in the CDCPUB1_SET change set being dropped. However, after all of the change tables contained within the change set have been dropped, any publisher can drop the CDCPUB1_SET change set with the DBMS_CDC_PUBLISH.DROP_CHANGE_SET subprogram.

Granting Privileges and Roles to the Publisher

Regardless of change data capture mode to be used, the staging database publisher must be granted the privileges and roles in the following list:

- EXECUTE_CATALOG_ROLE privilege
- SELECT_CATALOG_ROLE privilege
- CREATE TABLE and CREATE SESSION privileges
- EXECUTE on the DBMS_CDC_PUBLISH package

For asynchronous HotLog, Distributed HotLog, and AutoLog publishing, the staging database publisher must be configured as an Oracle Streams administrator and also be granted the CREATE SEQUENCE privilege, as follows. (See *Oracle Streams Concepts and Administration* for information on configuring an Oracle Streams administrator.)

- Be granted the CREATE SEQUENCE privilege
- Be granted the DBA role
- Be the GRANTEE specified in a DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE subprogram issued by the staging database DBA

For asynchronous Distributed HotLog publishing, the *source* database publisher must be granted the DBA role and must be the grantee specified in a DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE subprogram.

Creating a Default Tablespace for the Publisher

Oracle recommends that when creating the publisher account on a staging database, the DBA specify a default tablespace for the publisher; the publisher should use this tablespace for any change tables he or she creates.

Password Files and Setting the REMOTE_LOGIN_PASSWORDFILE Parameter

You must ensure that a password file has been generated. This occurs automatically when using the Database Configuration Assistant.

See Also:

- *Oracle Database Administrator's Guide*
- "Database Configuration Assistant Considerations" on page 17-72 for more information on setting passwords

Determining the Mode in Which to Capture Data

These factors influence the decision on the mode in which to capture change data:

- Whether or not the staging database is remote from the source database
- Tolerance for latency between changes made on the source database and changes captured by Change Data Capture. Note that latency generally increases from Synchronous to Asynchronous AutoLog Archive in Table 17–4
- Performance impact on the source database transactions and overall database performance
- Whether the source and staging databases will be running on the same hardware, using the same operating systems, or using the same Oracle database release

Table 17–4 summarizes these factors that influence the mode decision.

Table 17–4 Factors Influencing Choice of Change Data Capture Mode

Mode	Location of, Hardware, and Software on Staging Database	Capture Mechanism	Source Database Performance Impact
Synchronous	Location must be the same as the source database and therefore hardware, operating system, and Oracle database release are the same as source system.	Change data is automatically committed as part of the same transaction it reflects.	Adds overhead to source database transactions to perform change data capture.
Asynchronous HotLog	Location must be the same as the source database and therefore hardware, operating system, and Oracle database release are the same as source system.	Change data is captured from the current online redo log file. Change sets are populated automatically as new transactions are committed.	Minimal impact on source database transactions to perform supplemental logging. Additional source database overhead to perform change data capture.

Table 17–4 (Cont.) Factors Influencing Choice of Change Data Capture Mode

Mode	Location of, Hardware, and Software on Staging Database	Capture Mechanism	Source Database Performance Impact
Asynchronous Distributed HotLog	Location is remote from the source database. Hardware, operating system, and Oracle database release can be different from the source system.	Change data is captured from the current online redo log file. The change set is populated automatically as new committed transactions arrive on the staging database.	Minimal impact on source database transactions to perform supplemental logging. Some overhead on the source database is incurred when mining the online redo log files.
Asynchronous AutoLog Online	Location is remote from the source database. Hardware, operating system, and Oracle database release are the same as source system.	Change data is captured from the standby redo log files. The change set is populated automatically as new committed transactions arrive on the staging database.	Minimal impact on source database transactions to perform supplemental logging. Minimal source database overhead for redo transport services.
Asynchronous AutoLog Archive	Location is remote from the source database. Hardware, operating system, and Oracle database release are the same as source system.	Change data is captured from archived redo log files. Change sets are populated automatically as archived redo log files arrive on the staging database.	Minimal impact on source database transactions to perform supplemental logging. Minimal source database overhead for redo transport services.

Setting Initialization Parameters for Change Data Capture Publishing

Initialization parameters must be set on the source or staging database, or both, for Change Data Capture to succeed. Which parameters to set depend on the mode in which Change Data Capture is publishing change data, and on whether the parameters are being set on the source or staging database.

The following sections describe the database initialization parameter settings for each mode of Change Data Capture. Sometimes the DBA is directed to add a value to a current setting. (The DBA can use the SQL `SHOW PARAMETERS` statement to see the current value of a setting.)

See *Oracle Database Reference* for general information about these database initialization parameters and *Oracle Streams Concepts and Administration* for more information about the database initialization parameters set for asynchronous publishing.

Initialization Parameters for Synchronous Publishing

Set the `JAVA_POOL_SIZE` parameter as follows:

```
JAVA_POOL_SIZE = 50000000
```

Initialization Parameters for Asynchronous HotLog Publishing

Table 17–5 lists the source database initialization parameters and their recommended settings for asynchronous HotLog publishing.

Table 17–5 Source Database Initialization Parameters for Asynchronous HotLog Publishing

Parameter	Recommended Value
COMPATIBLE	11.0
JAVA_POOL_SIZE	50000000
JOB_QUEUE_PROCESSES	(current value) + 2
PARALLEL_MAX_SERVERS	(current value) + (5 * (the number of change sets planned))
PROCESSES	(current value) + (7 * (the number of change sets planned))
SESSIONS	(current value) + (2 * (the number of change sets planned))
STREAMS_POOL_SIZE	▪ If the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sets planned) * (21 MB)) ▪ If the current value of the STREAMS_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sets planned) * (21 MB)) See *Oracle Streams Concepts and Administration* for information on how the STREAMS_POOL_SIZE parameter is applied when changed dynamically.
UNDO_RETENTION	3600

Initialization Parameters for Asynchronous Distributed HotLog Publishing

Table 17–6 lists the source database initialization parameters and their recommended settings for asynchronous Distributed HotLog publishing when the source database is Oracle Database release 10.1.0, 10.2.0, 11.1.0, or 11.2.0.

Table 17–7 lists the source database initialization parameters and their recommended settings for asynchronous Distributed HotLog publishing when the source database is Oracle Database release 9.2.

Table 17–8 lists the staging database initialization parameters and their recommended settings for asynchronous Distributed HotLog publishing. These are the same regardless of which Oracle database release is being used for the source database.

Table 17–6 Source Database (10.1.0, 10.2.0, 11.1.0, 11.2.0) Initialization Parameters for Asynchronous Distributed HotLog Publishing

Parameter	Recommended Value
COMPATIBLE	11.0, depending on the source Oracle database release
GLOBAL_NAMES	TRUE
JOB_QUEUE_PROCESSES	(current value) + 2
OPEN_LINKS	4, or the number of Distributed HotLog change sources planned, whichever is greater.
PARALLEL_MAX_SERVERS	(current value) + (3 * (the number of change sources planned))
PROCESSES	(current value) + (4 * (the number of change sources planned))

Table 17–6 (Cont.) Source Database (10.1.0, 10.2.0, 11.1.0, 11.2.0) Initialization Parameters for Asynchronous Distributed HotLog Publishing

Parameter	Recommended Value
SESSIONS	(current value) + (the number of change sources planned)
STREAMS_POOL_SIZE	▪ If the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sources planned) * (20 MB)) ▪ If the current value of the STREAMS_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sets planned) * (20 MB)) See *Oracle Streams Concepts and Administration* for information on how the STREAMS_POOL_SIZE parameter is applied when changed dynamically.
UNDO_RETENTION	3600

Table 17–7 Source Database (9.2) Initialization Parameters for Asynchronous Distributed HotLog Publishing

Parameter	Recommended Value
COMPATIBLE	9.2.0
GLOBAL_NAMES	TRUE
JOB_QUEUE_PROCESSES	(current value) + 2
LOG_PARALLELISM	1
LOGMNR_MAX_PERSISTENT_SESSIONS	Value equal to the number of change sources planned
OPEN_LINKS	4, or the number of Distributed HotLog change sources planned, whichever is greater
PARALLEL_MAX_SERVERS	(current value) + (3 * (the number of change sources planned))
PROCESSES	(current value) + (4 * (the number of change sources planned))
SESSIONS	(current value) + (the number of change sources planned)
SHARED_POOL_SIZE	▪ If the current value of the SHARED_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sources planned) * (20 MB)) ▪ If the current value of the SHARED_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sources planned) * (20 MB))
UNDO_RETENTION	3600

Table 17–8 Staging Database (11.2.0) Initialization Parameters for Asynchronous Distributed HotLog Publishing

Parameter	Recommended Value
COMPATIBLE	11.0
GLOBAL_NAMES	TRUE
JAVA_POOL_SIZE	50000000
OPEN_LINKS	4, or the number of Distributed HotLog change sets planned, whichever is greater.
PARALLEL_MAX_SERVERS	(current value) + (2 * (the number of change sets planned))

Table 17–8 (Cont.) Staging Database (11.2.0) Initialization Parameters for Asynchronous Distributed HotLog Publishing

Parameter	Recommended Value
PROCESSES	(current value) + (3 * (the number of change sets planned))
SESSIONS	(current value) + (the number of change sets planned)
STREAMS_POOL_SIZE	▪ If the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sets planned) * (11 MB)) ▪ If the current value of the STREAMS_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sets planned) * (11 MB)) See *Oracle Streams Concepts and Administration* for information on how the STREAMS_POOL_SIZE parameter is applied when changed dynamically.

Initialization Parameters for Asynchronous AutoLog Publishing

Table 17–9 lists the database initialization parameters and their recommended settings for the asynchronous AutoLog publishing source database and Table 17–10 lists the database initialization parameters and their recommended settings for the asynchronous AutoLog publishing staging database.

Table 17–9 Source Database Initialization Parameters for Asynchronous AutoLog Publishing

Parameter	Recommended Value
COMPATIBLE	11.0
LOG_ARCHIVE_DEST_1[1]	The directory specification on the source database where its own archived redo log files are to be kept.
LOG_ARCHIVE_DEST_2[1]	This parameter must include the SERVICE, ARCH or LGWR ASYNC, OPTIONAL, NOREGISTER, and REOPEN attributes so that redo transport services are configured to copy the redo log files from the source database to the staging database. This parameter must also include either the VALID_FOR or the TEMPLATE attribute depending on the AutoLog option. These attributes are set as follows: ■ SERVICE specifies the network name of the staging database. ■ ARCH or LGWR ASYNC To use the AutoLog online option, specify LGWR ASYNC. LGWR ASYNC specifies that the log writer process (LGWR) copy redo data asynchronously to the staging database as the redo is generated on the source database. The copied redo data becomes available to Change Data Capture after its source database transaction commits. To use the AutoLog archive option, specify either ARCH or LGWR ASYNC. ARCH specifies that the archiver process (ARCn) copy the redo log files to the staging database after a source database log switch occurs. LGWR ASYNC specifies that the log writer process (LGWR) copy redo data asynchronously to the staging database as the redo is generated on the source database. For both ARCH and LGWR ASYNC, the copied redo data becomes available to Change Data Capture only after a source database log switch occurs when using the AutoLog archive option. ■ OPTIONAL specifies that the copying of a redo log file to the staging database need not succeed before the corresponding online redo log at the source database can be overwritten. This is needed to avoid stalling operations on the source database due to a transmission failure to the staging database. The original redo log file remains available to the source database in either archived or backed up form, if it is needed. ■ NOREGISTER specifies that the staging database location is not recorded in the staging database control file. ■ REOPEN specifies the minimum number of seconds the log writer process (LGWR) or the archive process (ARCn) should wait before trying to access the staging database if a previous attempt to access this location failed. ■ VALID_FOR When using the AutoLog online option, set VALID_FOR either to (ONLINE_LOGFILE, PRIMARY_ROLE) or (ONLINE_LOGFILE, ALL_ROLES) to enable redo data to be copied from the online redo log on the source database to the standby redo log at the staging database. ■ TEMPLATE When using the AutoLog archive option, specify TEMPLATE to define a directory specification and a format template for the file name used for the archived redo log files that are copied to the staging database.[2]
LOG_ARCHIVE_DEST_STATE_1[1]	ENABLE Indicates that redo transport services can transmit archived redo log files to this destination.
LOG_ARCHIVE_DEST_STATE_2[1]	ENABLE Indicates that redo transport services can transmit redo log files to this destination.
LOG_ARCHIVE_FORMAT[2]	"arch_%s_%t_%r.dbf" Specifies a format template for the default file name when archiving redo log files.[2] The string value (arch) and the file name extension (.dbf) do not have to be exactly as specified here.
REMOTE_ARCHIVE_ENABLE	TRUE Indicates that this source database can send redo log files to remote destinations.

[1] The integer value in this parameter can be any value between 1 and 10. In this manual, the values 1 and 2 are used. For each LOG_ARCHIVE_DEST_*n* parameter, there must be a corresponding LOG_ARCHIVE_DEST_STATE_*n* parameter that specifies the same value for *n*.

[2] In the format template, %t corresponds to the thread number, %s corresponds to the sequence number, and %r corresponds to the resetlogs ID. Together, these ensure that unique names are constructed for the copied redo log files. Each of these items must be present, but their ordering and format are flexible.

Table 17–10 Staging Database Initialization Parameters for Asynchronous AutoLog Publishing

Parameter	Recommended Value
COMPATIBLE	11.2.0
GLOBAL_NAMES	TRUE
JAVA_POOL_SIZE	50000000
LOG_ARCHIVE_DEST_1[1]	The directory specification on the staging database where its own archived redo log files are to be kept. If the staging database has an AutoLog online change source, the following attributes should be specified: ■ LOCATION specifies a unique directory path name for the staging database's own archived redo log files. ■ Set VALID_FOR either to (ONLINE_LOGFILE, PRIMARY_ROLE) or (ONLINE_LOGFILE, ALL_ROLES) to enable the online redo log file to be archived locally.
LOG_ARCHIVE_DEST_2[1]	If the staging database has an AutoLog online change source, this specifies the standby redo log files on the staging database that receive change data from the source database. It is very important to specify a unique location for these standby redo log files so that they do not overwrite the staging database's own archived log files. ■ LOCATION specifies a unique directory path name for the staging database's standby redo log files. ■ MANDATORY specifies that a standby redo log file must be successfully archived before it can be overwritten. ■ Set VALID_FOR either to (STANDBY_LOGFILE, PRIMARY_ROLE) or (STANDBY_LOGFILE, ALL_ROLES) to enable the staging database to receive change data from the source database and write it to the staging database standby log files.
LOG_ARCHIVE_DEST_STATE_1[1]	ENABLE Indicates that redo transport services can transmit archived redo log files to this destination.
LOG_ARCHIVE_DEST_STATE_2[1]	ENABLE Indicates that redo transport services can transmit redo log files to this destination.
LOG_ARCHIVE_FORMAT[2]	"arch_%s_%t_%r.dbf" Specifies a format template for the default file name when archiving redo log files[2]. The string value (arch) and the file name extension (.dbf) do not have to be exactly as specified here.
JOB_QUEUE_PROCESSES	2
PARALLEL_MAX_SERVERS	(current value) + (5 * (the number of change sets planned))
PROCESSES	(current value) + (7 * (the number of change sets planned))
REMOTE_ARCHIVE_ENABLE	TRUE Indicates that this staging database can receive remotely archived redo log files.

Table 17–10 (Cont.) Staging Database Initialization Parameters for Asynchronous AutoLog Publishing

Parameter	Recommended Value
SESSIONS	(current value)+ (2 * (the number of change sets planned))
STREAMS_POOL_SIZE	▪ If the current value of the STREAMS_POOL_SIZE parameter is 50 MB or greater, then set this parameter to: (current value) + ((the number of change sets planned) * (21 MB)) ▪ If the current value of the STREAMS_POOL_SIZE parameter is less than 50 MB, then set the value of this parameter to: 50 MB + ((the number of change sets planned) * (21 MB)) See *Oracle Streams Concepts and Administration* for information on how the STREAMS_POOL_SIZE parameter is applied when changed dynamically.
UNDO_RETENTION	3600

[1] The integer value in this parameter can be any value between 1 and 10. In this manual, the values 1 and 2 are used. For each LOG_ARCHIVE_DEST_*n* parameter, there must be a corresponding LOG_ARCHIVE_DEST_STATE_*n* parameter that specifies the same value for *n*.

[2] In the format template, %t corresponds to the thread number, %s corresponds to the sequence number, and %r corresponds to the resetlogs ID. Together, these ensure that unique names are constructed for the copied redo log files. Each of these items must be present, but their ordering and format are flexible.

Adjusting Initialization Parameter Values When Oracle Streams Values Change

Asynchronous Change Data Capture uses an Oracle Streams configuration for each change set. This Streams configuration consists of a Streams capture process and a Streams apply process, with an accompanying queue and queue table. Each Streams configuration uses additional processes, parallel execution servers, and memory. For details about the Streams architecture, see *Oracle Streams Concepts and Administration*.

If anything in your configuration changes, initialization parameters may need to be adjusted. See Table 17–10, " Staging Database Initialization Parameters for Asynchronous AutoLog Publishing" for more information.

Tracking Changes to the CDC Environment

You can track when partition maintenance operations, direct-path loads, and DDL operations occur in a change table. To do this, set the ddl_markers flag in the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure.

Publishing Change Data

The following sections provide step-by-step instructions on performing the various types of publishing:

- Performing Synchronous Publishing
- Performing Asynchronous HotLog Publishing
- Performing Asynchronous Distributed HotLog Publishing
- Performing Asynchronous AutoLog Publishing

Performing Synchronous Publishing

For synchronous Change Data Capture, the publisher must use the predefined change source, SYNC_SOURCE. The publisher can define new change sets or can use the predefined change set, SYNC_SET. The publisher must not create change tables on

source tables owned by SYS or SYSTEM because triggers do not fire and therefore changes are not captured.

This example shows how to create a change set. If the publisher wants to use the predefined SYNC_SET, he or she should skip Step 3 and specify SYNC_SET as the change set name in the remaining steps.

This example assumes that the publisher and the source database DBA are two different people.

Note that for synchronous Change Data Capture, the source database and the staging database are the same.

Step 1 Source Database DBA: Set the JAVA_POOL_SIZE parameter.

The source database DBA sets the database initialization parameters, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20.

```
java_pool_size = 50000000
```

Step 2 Source Database DBA: Create and grant privileges to the publisher.

The source database DBA creates a user (for example, cdcpub), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform the operations needed to create Change Data Capture change sets and change tables on the source database, as described in "Creating a User to Serve As a Publisher" on page 17-18. This example assumes that the tablespace ts_cdcpub has already been created.

```
CREATE USER cdcpub IDENTIFIED EXTERNALLY DEFAULT TABLESPACE ts_cdcpub
QUOTA UNLIMITED ON SYSTEM
QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO cdcpub;
GRANT CREATE TABLE TO cdcpub;
GRANT CREATE TABLESPACE TO cdcpub;
GRANT CREATE JOB TO cdcpub;
GRANT UNLIMITED TABLESPACE TO cdcpub;
GRANT SELECT_CATALOG_ROLE TO cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO cdcpub;
GRANT ALL ON sh.sales TO cdcpub;
GRANT ALL ON sh.products TO cdcpub;
GRANT EXECUTE ON DBMS_CDC_PUBLISH TO cdcpub;
```

Step 3 Staging Database Publisher: Create a change set.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create change sets.

The following example shows how to create a change set called CHICAGO_DAILY:

```
BEGIN
    DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
    change_set_name    => 'CHICAGO_DAILY',
    description        => 'Change set for sales history info',
    change_source_name => 'SYNC_SOURCE');
END;
/
```

The change set captures changes from the predefined change source SYNC_SOURCE. Because begin_date and end_date parameters cannot be specified for synchronous change sets, capture begins at the earliest available change data and continues capturing change data indefinitely.

Step 4 Staging Database Publisher: Create a change table.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to create change tables.

The publisher can set the options_string field of the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to have more control over the physical properties and tablespace properties of the change table. The options_string field can contain any option, except partitioning, that is available in the CREATE TABLE statement.

The following example creates a change table that captures changes that occur on a source table. The example uses the sample schema table sh.products as the source table. It assumes that the publisher has already created the TS_CHICAGO_DAILY tablespace.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
   owner             => 'cdcpub',
   change_table_name => 'products_ct',
   change_set_name   => 'CHICAGO_DAILY',
   source_schema     => 'SH',
   source_table      => 'PRODUCTS',
   column_type_list  => 'PROD_ID NUMBER(6),
                         PROD_NAME VARCHAR2(50),
                         PROD_LIST_PRICE NUMBER(8,2)',
   capture_values    => 'both',
   rs_id             => 'y',
   row_id            => 'n',
   user_id           => 'n',
   timestamp         => 'n',
   object_id         => 'n',
   source_colmap     => 'y',
   target_colmap     => 'y',
   options_string    => 'TABLESPACE TS_CHICAGO_DAILY');
END;
 /
```

This statement creates a change table named products_ct within the change set CHICAGO_DAILY. The column_type_list parameter identifies the columns captured by the change table. The source_schema and source_table parameters identify the schema and source table that reside in the source database.

The capture_values setting in the example indicates that for update operations, the change data contains two separate rows for each row that changed: one row contains the row values before the update occurred, and the other row contains the row values after the update occurred.

See "Managing Change Tables" on page 17-54 for more information.

Step 5 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the SELECT privilege on change tables for users and roles. The publisher grants access to specific change tables. Without this step, a subscriber cannot access any change data. This example assumes that user subscriber1 already exists.

```
GRANT SELECT ON cdcpub.products_ct TO subscriber1;
```

The Change Data Capture synchronous system is now ready for subscriber1 to create subscriptions.

Performing Asynchronous HotLog Publishing

Change Data Capture uses Oracle Streams local capture to perform asynchronous HotLog publishing. See *Oracle Streams Concepts and Administration* for information on Streams local capture.

For HotLog Change Data Capture, the publisher must use the predefined change source, HOTLOG_SOURCE, and must create the change sets and the change tables that contain the changes. The staging database is always the source database. This example assumes that the publisher and the source database DBA are two different people.

Note that for asynchronous HotLog Change Data Capture, the source database and the staging database are the same.

The following steps set up redo logging, Oracle Streams, and Change Data Capture for asynchronous HotLog publishing:

Step 1 Source Database DBA: Set the database initialization parameters.

The source database DBA sets the database initialization parameters, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20. In this example, one change set is defined and the current value of the STREAMS_POOL_SIZE parameter is 50 MB.

```
compatible = 11.0
java_pool_size = 50000000
job_queue_processes = 2
parallel_max_servers = <current value> + 5
processes = <current value> + 7
sessions = <current value> + 2
streams_pool_size = <current value> + 21 MB
undo_retention = 3600
```

Step 2 Source Database DBA: Alter the source database.

The source database DBA performs the following three tasks. The second is required. The first and third are optional, but recommended. It is assumed that the database is currently running in ARCHIVELOG mode.

1. Place the database into FORCE LOGGING logging mode to protect against unlogged direct write operations in the source database that cannot be captured by asynchronous Change Data Capture:

   ```
   ALTER DATABASE FORCE LOGGING;
   ```

   Note that logging can also be enforced at the tablespace or at the table level.

2. Enable supplemental logging. Supplemental logging places additional column data into a redo log file whenever an UPDATE operation is performed. Minimally, database-level minimal supplemental logging must be enabled for any Change Data Capture source database:

   ```
   ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
   ```

3. Create an unconditional log group on all columns to be captured in the source table. Source table columns that are unchanged and are not in an unconditional log group, will be null in the change table, instead of reflecting their actual source table values. (This example captures rows in the sh.products table only. The source database DBA would repeat this step for each source table for which change tables will be created.)

   ```
   ALTER TABLE sh.products
   ADD SUPPLEMENTAL LOG GROUP log_group_products
   ```

```
(PROD_ID, PROD_NAME, PROD_LIST_PRICE) ALWAYS;
```

If you intend to capture all the column values in a row whenever a column in that row is updated, you can use the following statement instead of listing each column one-by-one in the ALTER TABLE statement. However, do not use this form of the ALTER TABLE statement if all columns are not needed. Logging all columns incurs more overhead than logging selected columns.

```
ALTER TABLE sh.products ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

See *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode and for information on FORCE LOGGING mode; see "Asynchronous Change Data Capture and Supplemental Logging" on page 17-70 and *Oracle Database Utilities* for more information about supplemental logging.

Step 3 Source Database DBA: Create and grant privileges to the publisher.

The source database DBA creates a user, (for example, cdcpub), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform the underlying Oracle Streams operations needed to create Change Data Capture change sets and change tables on the source database, as described in "Creating a User to Serve As a Publisher" on page 17-18. This example assumes that the ts_cdcpub tablespace has already been created.

```
CREATE USER cdcpub IDENTIFIED EXTERNALLY DEFAULT TABLESPACE ts_cdcpub
QUOTA UNLIMITED ON SYSTEM
QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO cdcpub;
GRANT CREATE TABLE TO cdcpub;
GRANT CREATE TABLESPACE TO cdcpub;
GRANT UNLIMITED TABLESPACE TO cdcpub;
GRANT SELECT_CATALOG_ROLE TO cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO cdcpub;
GRANT CREATE SEQUENCE TO cdcpub;
GRANT DBA TO cdcpub;
GRANT EXECUTE on DBMS_CDC_PUBLISH TO cdcpub;

EXECUTE DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE(GRANTEE => 'cdcpub');
```

Note that for HotLog Change Data Capture, the source database and the staging database are the same database.

Step 4 Source Database DBA: Prepare the source tables.

The source database DBA must prepare the source tables on the source database for asynchronous Change Data Capture by instantiating each source table. Instantiating each source table causes the underlying Oracle Streams environment to record the information it needs to capture each source table's changes. The source table structure and the column datatypes must be supported by Change Data Capture. See "Datatypes and Table Structures Supported for Asynchronous Change Data Capture" on page 17-71 for more information.

```
BEGIN
DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(TABLE_NAME => 'sh.products');
END;
/
```

Step 5 Staging Database Publisher: Create change sets.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create change sets. Note that when Change Data Capture

creates a change set, its associated Oracle Streams capture and apply processes are also created (but not started).

The following example creates a change set called CHICAGO_DAILY that captures changes starting today, and stops capturing change data 5 days from now.

```
BEGIN
       DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       description => 'Change set for product info',
       change_source_name => 'HOTLOG_SOURCE',
       stop_on_ddl => 'y',
       begin_date => sysdate,
       end_date => sysdate+5);
END;
/
```

The change set captures changes from the predefined HOTLOG_SOURCE change source.

Step 6 Staging Database Publisher: Create the change tables that will contain the changes to the source tables.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure on the staging database to create change tables.

The publisher creates one or more change tables for each source table to be published, specifies which columns should be included, and specifies the combination of before and after images of the change data to capture.

The following example creates a change table on the staging database that captures changes made to a source table on the source database. The example uses the sample table sh.products as the source table.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
   owner              => 'cdcpub',
   change_table_name  => 'products_ct',
   change_set_name    => 'CHICAGO_DAILY',
   source_schema      => 'SH',
   source_table       => 'PRODUCTS',
   column_type_list   => 'PROD_ID NUMBER(6), PROD_NAME VARCHAR2(50),
      PROD_LIST_PRICE NUMBER(8,2)',
   capture_values     => 'both',
   rs_id              => 'y',
   row_id             => 'n',
   user_id            => 'n',
   timestamp          => 'n',
   object_id          => 'n',
   source_colmap      => 'n',
   target_colmap      => 'y',
   options_string     => 'TABLESPACE TS_CHICAGO_DAILY');
END;
/
```

This statement creates a change table named products_ct within change set CHICAGO_DAILY. The column_type_list parameter identifies the columns to be captured by the change table. The source_schema and source_table parameters identify the schema and source table that reside on the source database.

The capture_values setting in this statement indicates that for update operations, the change data will contain two separate rows for each row that changed: one row

will contain the row values before the update occurred and the other row will contain the row values after the update occurred.

The `options_string` parameter in this statement specifies a tablespace for the change table. (This example assumes that the publisher previously created the `TS_CHICAGO_DAILY` tablespace.)

See "Managing Change Tables" on page 17-54 for more information.

Step 7 Staging Database Publisher: Enable the change set.

Because asynchronous change sets are always disabled when they are created, the publisher must alter the change set to enable it. The Oracle Streams capture and apply processes are started when the change set is enabled.

```
BEGIN
   DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
      change_set_name => 'CHICAGO_DAILY',
      enable_capture => 'y');
END;
/
```

Step 8 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the `SELECT` privilege on change tables for users and roles. The publisher grants access to specific change tables. Without this step, a subscriber cannot access change data. This example assumes that user `subscriber1` already exists.

```
GRANT SELECT ON cdcpub.products_ct TO subscriber1;
```

The Change Data Capture asynchronous HotLog system is now ready for `subscriber1` to create subscriptions.

Performing Asynchronous Distributed HotLog Publishing

For Distributed HotLog Change Data Capture, the staging database is remote from the source database. However, steps must be performed on both the source database and the staging database to set up database links, Oracle Streams, and Change Data Capture. Tasks must be performed by a source database DBA, a staging database DBA, and a publisher on each database, as follows:

- The source database DBA sets up Oracle Net, database initialization parameters, alters the source database to enable force logging and supplemental logging, and creates the source database publisher.
- The staging database DBA sets database initialization parameters and creates a staging database publisher.
- The source database publisher sets up a database link from the source database to the staging database.
- The staging database publisher sets up a database link from the staging database to the source database, creates new change sources, change sets, and the change tables that will contain the changes that are made to individual source tables, and grants necessary privileges to subscribers.

This example assumes that the source database DBA, the staging database DBA, and the publishers are four different people.

Step 1 Source Database DBA: Prepare the source database.

The source database DBA performs the following tasks:

1. Configures Oracle Net so that the source database can communicate with the staging database. See *Oracle Database Net Services Administrator's Guide* for information about Oracle Net.
2. Sets the database initialization parameters on the source database as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20. In the following code example, the source database is Oracle Database release 11.2, the number of planned change sources is 1, and the current value of the STREAMS_POOL_SIZE parameter is 50 MB:

```
compatible = 11.0
global_names = true
job_queue_processes = <current value> + 2
open_links = 4
parallel_max_servers = <current value> + 3
processes = <current value> + 4
sessions = <current value> + 1
streams_pool_size = <current value> + 20 MB
undo_retention = 3600
```

Step 2 Staging Database DBA: Set the database initialization parameters.

The staging database DBA performs the following tasks:

1. Sets the database initialization parameters on the staging database, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20. In this example, one change set will be defined and the current value for the STREAMS_POOL_SIZE parameter is 50 MB or greater:

```
compatible = 11.0
global_names = true
java_pool_size = 50000000
open_links = 4
job_queue_processes = 2
parallel_max_servers = <current_value> + 2
processes = <current_value> + 3
sessions = <current value> + 1
streams_pool_size = <current_value> + 11 MB
undo_retention = 3600
```

Step 3 Source Database DBA: Alter the source database.

The source database DBA performs the following three tasks. The second is required. The first and third are optional, but recommended. It is assumed that the database is currently running in ARCHIVELOG mode.

1. Place the database into FORCE LOGGING logging mode to protect against unlogged direct writes in the source database that cannot be captured by asynchronous Change Data Capture:

```
ALTER DATABASE FORCE LOGGING;
```

2. Enable supplemental logging. Supplemental logging places additional column data into a redo log file whenever an update operation is performed.

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

3. Create an unconditional log group on all columns to be captured in the source table. Source table columns that are unchanged and are not in an unconditional

log group, will be null in the change table, instead of reflecting their actual source table values. (This example captures rows in the `sh.products` table only. The source database DBA would repeat this step for each source table for which change tables will be created).

```
ALTER TABLE sh.products
ADD SUPPLEMENTAL LOG GROUP log_group_products
    (PROD_ID, PROD_NAME, PROD_LIST_PRICE) ALWAYS;
```

If you intend to capture all the column values in a row whenever a column in that row is updated, you can use the following statement instead of listing each column one-by-one in the `ALTER TABLE` statement. However, do not use this form of the `ALTER TABLE` statement if all columns are not needed. Logging all columns incurs more overhead than logging selected columns.

```
ALTER TABLE sh.products ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

See Also:

- *Oracle Database Administrator's Guide* for information about running a database in `ARCHIVELOG` mode
- "Asynchronous Change Data Capture and Supplemental Logging" on page 17-70 for more information on supplemental logging
- *Oracle Database Utilities* for more information on supplemental logging

Step 4 Source Database DBA: Create and grant privileges to the publisher.

The source database DBA creates a user, (for example, `source_cdcpub`), to serve as the source database publisher and grants the necessary privileges to the publisher so that he or she can set up a database link from the source database to connect to the staging database publisher, as described in "Creating a User to Serve As a Publisher" on page 17-18. For example:

```
CREATE USER source_cdcpub IDENTIFIED EXTERNALLY
   QUOTA UNLIMITED ON SYSTEM
   QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO source_cdcpub;
GRANT DBA TO source_cdcpub;
GRANT CREATE DATABASE LINK TO source_cdcpub;
GRANT EXECUTE on DBMS_CDC_PUBLISH TO source_cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO source_cdcpub;
GRANT SELECT_CATALOG_ROLE TO source_cdcpub;
EXECUTE DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE(
      GRANTEE=> 'source_cdcpub');
```

Step 5 Staging Database DBA: Create and grant privileges to the publisher.

The staging database DBA creates a user, (for example, `staging_cdcpub`), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform underlying Oracle Streams operations, create a database link from the staging database to connect to the source database publisher, create the change sources on the source database, and the change sets and change tables on the staging database, as described in "Creating a User to Serve As a Publisher" on page 17-18. For example:

```
CREATE USER staging_cdcpub IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE ts_cdcpub
   QUOTA UNLIMITED ON SYSTEM;
GRANT CREATE SESSION TO staging_cdcpub;
```

```
GRANT CREATE TABLE TO staging_cdcpub;
GRANT CREATE TABLESPACE TO staging_cdcpub;
GRANT UNLIMITED TABLESPACE TO staging_cdcpub;
GRANT SELECT_CATALOG_ROLE TO staging_cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO staging_cdcpub;
GRANT CONNECT, RESOURCE, DBA TO staging_cdcpub;
GRANT CREATE SEQUENCE TO staging_cdcpub;
EXECUTE DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE(grantee => 'staging_cdcpub');
GRANT CREATE DATABASE LINK TO staging_cdcpub;
```

Step 6 Source Database Publisher: Create Database Link

The source database publisher creates a link from the source database to the staging database. Because the `GLOBAL_NAMES` initialization parameter is set to `TRUE`, the database link name will be same as that of the staging database. It connects to the staging database using the username and password of the staging database publisher. In this example, the name of the staging database is `staging_db`:

```
CREATE DATABASE LINK staging_db
CONNECT TO staging_cdcpub IDENTIFIED BY Stg395V3
USING 'staging_db';
```

For detailed information on database links, see *Oracle Database Administrator's Guide*.

Step 7 Staging Database Publisher: Create Database Link

The staging database publisher creates a link from the staging database to the source database. Again, because the `GLOBAL_NAMES` initialization parameter is set to `TRUE`, the database link name will be the same as that of the source database. It connects to the source database using the username and password of the source database publisher. In this example, the name of the source database is `source_db`:

```
CREATE DATABASE LINK source_db
CONNECT TO source_cdcpub IDENTIFIED BY Lvh412A7
USING 'source_db';
```

Note that this database link must exist when creating, altering or dropping Distributed HotLog change sources, change sets and change tables. However, this database link is not required for change capture to occur. Once the required Distributed HotLog change sources, change sets and change tables are in place and enabled, this database link can be dropped without interrupting change capture. This database link would need to be re-created to create, alter or drop Distributed HotLog change sources, change sets and change tables.

Step 8 Staging Database Publisher: Identify the change source database and create the change sources.

The staging database publisher uses the `DBMS_CDC_PUBLISH.CREATE_HOTLOG_CHANGE_SOURCE` procedure on the staging database to create the Distributed HotLog change sources on the source database.

Note that when Change Data Capture creates a change source, its associated Oracle Streams capture process is also created (but not started).

The publisher creates the Distributed HotLog change source and specifies the database link defined from the staging database to the source database. The name of the database link is the same as the name of the source database:

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_HOTLOG_CHANGE_SOURCE(
       change_source_name => 'CHICAGO',
```

```
      description => 'test source',
      source_database => 'source_db');
END;
/
```

Step 9 Staging Database Publisher: Create change sets.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create a change set. A Distributed HotLog change source can contain one or more change sets on the same staging database. The publisher cannot provide beginning and ending dates.

Note that when Change Data Capture creates a change set, its associated Oracle Streams apply process is also created (but not started).

The following example shows how to create a change set called CHICAGO_DAILY that captures changes starting today, and continues capturing change data indefinitely. (If, at some time in the future, the publisher decides that he or she wants to stop capturing change data for this change set, he or she should disable the change set and then drop it.)

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       description => 'change set for product info',
       change_source_name => 'CHICAGO',
       stop_on_ddl => 'y');
END;
/
```

Step 10 Staging Database Publisher: Create the change tables.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure on the staging database to create change tables.

The publisher creates one or more change tables for each source table to be published, specifies which columns should be included, and specifies the combination of before and after images of the change data to capture.

The publisher can set the options_string field of the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to have more control over the physical properties and tablespace properties of the change tables. The options_string field can contain any option available (except partitioning) on the CREATE TABLE statement. In this example, it specifies a tablespace for the change set. (This example assumes that the publisher previously created the TS_CHICAGO_DAILY tablespace.)

The following example creates a change table on the staging database that captures changes made to a source table in the source database. The example uses the sample table sh.products.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
       owner             => 'staging_cdcpub',
       change_table_name => 'products_ct',
       change_set_name   => 'CHICAGO_DAILY',
       source_schema     => 'SH',
       source_table      => 'PRODUCTS',
       column_type_list  => 'PROD_ID NUMBER(6), PROD_NAME VARCHAR2(50),
          PROD_LIST_PRICE NUMBER(8,2),
          JOB_ID VARCHAR2(10), DEPARTMENT_ID NUMBER(4)',
       capture_values    => 'both',
       rs_id             => 'y',
```

```
       row_id            => 'n',
       user_id           => 'n',
       timestamp         => 'n',
       object_id         => 'n',
       source_colmap     => 'n',
       target_colmap     => 'y',
       options_string    => 'TABLESPACE TS_CHICAGO_DAILY');
END;
/
```

This example creates a change table named `products_ct` within change set `CHICAGO_DAILY`. The `column_type_list` parameter identifies the columns captured by the change table. The `source_schema` and `source_table` parameters identify the schema and source table that reside in the source database, not the staging database.

The `capture_values` setting in the example indicates that for update operations, the change data will contain two separate rows for each row that changed: one row will contain the row values before the update occurred and the other row will contain the row values after the update occurred. Note that for Distributed HotLog change sets, the `object_id` and `source_colmap` capture values must be set to `'n'`. If the change source is 9.2 or 10.1, `row_id` and `user_id` must also be `'n'`.

See "Managing Change Tables" on page 17-54 for more information.

Step 11 Staging Database Publisher: Enable the change source.

Because Distributed HotLog change sources are always disabled when they are created, the publisher must alter the change source to enable it. The Oracle Streams capture process is started when the change source is enabled.

```
BEGIN
  DBMS_CDC_PUBLISH.ALTER_HOTLOG_CHANGE_SOURCE(
    change_source_name => 'CHICAGO',
    enable_source      => 'Y');
END;
/
```

Step 12 Staging Database Publisher: Enable the change set.

Because asynchronous change sets are always disabled when they are created, the publisher must alter the change set to enable it. The Oracle Streams apply processes is started when the change set is enabled.

```
BEGIN
   DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       enable_capture  => 'y');
END;
/
```

Step 13 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the `SELECT` privilege on change tables for users and roles on the staging database. The publisher grants access to specific change tables. Without this step, a subscriber cannot access any change data. This example assumes that user `subscriber1` already exists.

```
GRANT SELECT ON staging_cdcpub.products_ct TO subscriber1;
```

The Change Data Capture Distributed HotLog system is now ready for `subscriber1` to create subscriptions.

Performing Asynchronous AutoLog Publishing

Change Data Capture uses Oracle Streams downstream capture to perform asynchronous AutoLog publishing. The Change Data Capture staging database is considered a downstream database in the Streams environment. The asynchronous AutoLog online option uses Streams real-time downstream capture. The asynchronous AutoLog archive option uses Streams archived-log downstreams capture. See *Oracle Streams Concepts and Administration* for information on Streams downstream capture.

For asynchronous AutoLog Change Data Capture, the publisher creates new change sources, as well as the change sets and the change tables that will contain the changes that are made to individual source tables.

Steps must be performed on both the source database and the staging database to set up redo transport services, Streams, and Change Data Capture for asynchronous AutoLog publishing. Because the source database and staging database are usually on separate systems, this example assumes that the source database DBA, the staging database DBA, and the publisher are different people.

Step 1 Source Database DBA: Prepare to copy redo log files from the source database.

The source database DBA and the staging database DBA must set up redo transport services to copy redo log files from the source database to the staging database and to prepare the staging database to receive these redo log files, as follows:

1. The source database DBA configures Oracle Net so that the source database can communicate with the staging database. See *Oracle Database Net Services Administrator's Guide* for information about Oracle Net.
2. The source database DBA sets the database initialization parameters on the source database as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20. In the following code examples, `STAGINGDB` is the network name of the staging database.

   The following is an example for the AutoLog online option:

   ```
   compatible = 11.0
   log_archive_dest_1 ="location=/oracle/dbs mandatory reopen=5"
   log_archive_dest_2 ="service=stagingdb lgwr async optional noregister reopen=5
                        valid_for=(online_logfile,primary_role)"
   log_archive_dest_state_1 = enable
   log_archive_dest_state_2 = enable
   log_archive_format="arch_%s_%t_%r.dbf"
   ```

   The following is an example for the AutoLog archive option:

   ```
   compatible = 11.0
   log_archive_dest_1="location=/oracle/dbs mandatory reopen=5"
   log_archive_dest_2 = "service=stagingdb arch optional noregister reopen=5
                         template=/usr/oracle/dbs/arch_%s_%t_%r.dbf"
   log_archive_dest_state_1 = enable
   log_archive_dest_state_2 = enable
   log_archive_format="arch_%s_%t_%r.dbf"
   ```

   See *Oracle Data Guard Concepts and Administration* for information on redo transport services.

Step 2 Staging Database DBA: Set the database initialization parameters.

The staging database must be run in `ARCHIVELOG` mode for Asynchronous Autolog CDC. The staging database DBA sets the database initialization parameters on the

staging database, as described in "Setting Initialization Parameters for Change Data Capture Publishing" on page 17-20. In these examples, one change set will be defined and the current value for the STREAMS_POOL_SIZE is 50 MB or greater.

The following is an example for the AutoLog online option:

```
compatible = 11.0
global_names = true
java_pool_size = 50000000
log_archive_dest_1="location=/oracle/dbs mandatory reopen=5
                    valid_for=(online_logfile,primary_role)"
log_archive_dest_2="location=/usr/oracle/dbs mandatory
                    valid_for=(standby_logfile,primary_role)"
log_archive_dest_state_1=enable
log_archive_dest_state_2=enable
log_archive_format="arch_%s_%t_%r.dbf"
job_queue_processes = 2
parallel_max_servers = <current_value> + 5
processes = <current_value> + 7
sessions = <current value> + 2
streams_pool_size = <current_value> + 21 MB
undo_retention = 3600
```

The following is an example for the AutoLog archive option:

```
compatible = 11.0
global_names = true
java_pool_size = 50000000
log_archive_dest_1="location=/oracle/dbs mandatory reopen=5
                    valid_for=(online_logfile,primary_role)"
log_archive_dest_2="location=/oracle/stdby mandatory
                    valid_for=(standby_logfile,primary_role)"
log_archive_dest_state_1=enable
log_archive_dest_state_2=enable
log_archive_format="arch_%s_%t_%r.dbf"
job_queue_processes = 2
parallel_max_servers = <current_value> + 5
processes = <current_value> + 7
sessions = <current value> + 2
streams_pool_size = <current_value> + 21 MB
undo_retention = 3600
```

Step 3 Source Database DBA: Alter the source database.

The source database DBA performs the following three tasks. The second is required. The first and third are optional, but recommended. It is assumed that the database is currently running in ARCHIVELOG mode.

1. Place the database into FORCE LOGGING logging mode to protect against unlogged direct writes in the source database that cannot be captured by asynchronous Change Data Capture:

   ```
   ALTER DATABASE FORCE LOGGING;
   ```

2. Enable supplemental logging. Supplemental logging places additional column data into a redo log file whenever an update operation is performed.

   ```
   ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
   ```

3. Create an unconditional log group on all columns to be captured in the source table. Source table columns that are unchanged and are not in an unconditional log group, will be null in the change table, instead of reflecting their actual source

table values. (This example captures rows in the sh.products table only. The source database DBA would repeat this step for each source table for which change tables will be created).

```
ALTER TABLE sh.products
ADD SUPPLEMENTAL LOG GROUP log_group_products
    (PROD_ID, PROD_NAME, PROD_LIST_PRICE) ALWAYS;
```

If you intend to capture all the column values in a row whenever a column in that row is updated, you can use the following statement instead of listing each column one-by-one in the ALTER TABLE statement. However, do not use this form of the ALTER TABLE statement if all columns are not needed. Logging all columns incurs more overhead than logging selected columns.

```
ALTER TABLE sh.products ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

See *Oracle Database Administrator's Guide* for information about running a database in ARCHIVELOG mode. See "Asynchronous Change Data Capture and Supplemental Logging" on page 17-70 and *Oracle Database Utilities* for more information on supplemental logging.

Step 4 Staging Database DBA: Create Standby Redo Log Files

This step is only needed for the AutoLog online option.

The staging database DBA must create the actual standby redo log files on the staging database:

1. Determine the log file size used on the source database because the standby redo log files must be the same size or larger. You can query V$LOG on the source database to determine the source database log file size.
2. Determine the number of standby log file groups required on the staging database. This must be at least one more than the number of online log file groups on the source database. You can query V$LOG on the source database to determine the number of online log file groups on the source database.
3. Use the SQL statement ALTER DATABASE ADD STANDBY LOGFILE to add the standby log file groups to the staging database:

   ```
   ALTER DATABASE ADD STANDBY LOGFILE GROUP 3
       ('/oracle/dbs/slog3a.rdo', '/oracle/dbs/slog3b.rdo') SIZE 500M;
   ```

   You can query V$STANDBY_LOG on the staging database to ensure that the standby log file groups have been added successfully.

Step 5 Staging Database DBA: Create and grant privileges to the publisher.

The staging database DBA creates a user, (for example, cdcpub), to serve as the Change Data Capture publisher and grants the necessary privileges to the publisher so that he or she can perform the underlying Oracle Streams operations needed to create Change Data Capture change sources, change sets, and change tables on the staging database, as described in "Creating a User to Serve As a Publisher" on page 17-18. For example:

```
CREATE USER cdcpub IDENTIFIED EXTERNALLY DEFAULT TABLESPACE ts_cdcpub
   QUOTA UNLIMITED ON SYSTEM
   QUOTA UNLIMITED ON SYSAUX;
GRANT CREATE SESSION TO cdcpub;
GRANT CREATE TABLE TO cdcpub;
GRANT CREATE TABLESPACE TO cdcpub;
GRANT UNLIMITED TABLESPACE TO cdcpub;
```

```
GRANT SELECT_CATALOG_ROLE TO cdcpub;
GRANT EXECUTE_CATALOG_ROLE TO cdcpub;
GRANT DBA TO cdcpub;
GRANT CREATE SEQUENCE TO cdcpub;
GRANT EXECUTE on DBMS_CDC_PUBLISH TO cdcpub;
EXECUTE DBMS_STREAMS_AUTH.GRANT_ADMIN_PRIVILEGE(grantee => 'cdcpub');
```

Step 6 Source Database DBA: Build the LogMiner data dictionary.

The source database DBA builds a LogMiner data dictionary at the source database so that redo transport services can transport this data dictionary to the staging database. This LogMiner data dictionary build provides the table definitions as they were just prior to beginning to capture change data. Change Data Capture automatically updates the data dictionary with any source table data definition language (DDL) operations that are made during the course of change data capture to ensure that the dictionary is always synchronized with the source database tables.

When building the LogMiner data dictionary, the source database DBA should get the SCN value of the data dictionary build. In Step 8, when the publisher creates a change source, he or she will need to provide this value as the `first_scn` parameter.

```
SET SERVEROUTPUT ON
VARIABLE f_scn NUMBER;
BEGIN
    :f_scn := 0;
    DBMS_CAPTURE_ADM.BUILD(:f_scn);
    DBMS_OUTPUT.PUT_LINE('The first_scn value is ' || :f_scn);
END;
/
The first_scn value is 207722
```

For asynchronous AutoLog publishing to work, it is critical that the source database DBA build the data dictionary (and the build completes) before the source tables are prepared. The source database DBA must be careful to follow Step 6 and Step 7 in the order they are presented here.

See *Oracle Streams Concepts and Administration* for more information on the LogMiner data dictionary.

Step 7 Source Database DBA: Prepare the source tables.

The source database DBA must prepare the source tables on the source database for asynchronous Change Data Capture by instantiating each source table so that the underlying Oracle Streams environment records the information it needs to capture each source table's changes. The source table structure and the column datatypes must be supported by Change Data Capture. See "Datatypes and Table Structures Supported for Asynchronous Change Data Capture" on page 17-71 for more information.

```
BEGIN
   DBMS_CAPTURE_ADM.PREPARE_TABLE_INSTANTIATION(
       TABLE_NAME => 'sh.products');
END;
/
```

Step 8 Source Database DBA: Get the global name of the source database.

In Step 8, the publisher will need to reference the global name of the source database. The global name of the source database will be used on the staging database to create the AutoLog change source. The source database DBA can query the `GLOBAL_NAME`

column in the GLOBAL_NAME view on the source database to retrieve this information for the publisher:

```
SELECT GLOBAL_NAME FROM GLOBAL_NAME;
GLOBAL_NAME
--------------------------------------------------------------------------------
HQDB
```

Step 9 Staging Database Publisher: Identify each change source database and create the change sources.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_AUTOLOG_CHANGE_SOURCE procedure on the staging database to create change sources.

The process of managing the capture system begins with the creation of a change source. A change source describes the source database from which the data will be captured, and manages the relationship between the source database and the staging database. A change source always specifies the SCN of a data dictionary build from the source database as its first_scn parameter.

The publisher gets the SCN of the data dictionary build and the global database name from the source database DBA (as shown in Step 5 and Step 7, respectively). If the publisher cannot get the value to use for the first_scn parameter value from the source database DBA, then, with the appropriate privileges, he or she can query the V$ARCHIVED_LOG view on the source database to determine the value. This is described in the DBMS_CDC_PUBLISH chapter of the *Oracle Database PL/SQL Packages and Types Reference*.

On the staging database, the publisher creates the AutoLog change source and specifies the global name as the source_database parameter value and the SCN of the data dictionary build as the first_scn parameter value. To create an AutoLog online change source:

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_AUTOLOG_CHANGE_SOURCE(
       change_source_name  => 'CHICAGO',
       description         => 'test source',
       source_database     => 'HQDB',
       first_scn           => 207722,
       online_log          => 'y');
END;
/
```

To create an AutoLog archive change source:

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_AUTOLOG_CHANGE_SOURCE(
       change_source_name   => 'CHICAGO',
       description          => 'test source',
       source_database      => 'HQDB',
       first_scn            => 207722);
END;
/
```

Step 10 Staging Database Publisher: Create change sets.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_SET procedure on the staging database to create change sets. The publisher can optionally provide beginning and ending dates to indicate where to begin and end the data capture.

Note that when Change Data Capture creates a change set, its associated Oracle Streams capture and apply processes are also created (but not started).

The following example shows how to create a change set called CHICAGO_DAILY that captures changes starting today, and continues capturing change data indefinitely. (If, at some time in the future, the publisher decides that he or she wants to stop capturing change data for this change set, he or she should disable the change set and then drop it.)

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
       change_set_name    => 'CHICAGO_DAILY',
       description        => 'change set for product info',
       change_source_name => 'CHICAGO',
       stop_on_ddl        => 'y');
END;
/
```

Step 11 Staging Database Publisher: Create the change tables.

The publisher uses the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure on the staging database to create change tables.

The publisher creates one or more change tables for each source table to be published, specifies which columns should be included, and specifies the combination of before and after images of the change data to capture.

The publisher can set the options_string field of the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure to have more control over the physical properties and tablespace properties of the change tables. The options_string field can contain any option available (except partitioning) on the CREATE TABLE statement. In this example, it specifies a tablespace for the change set. (This example assumes that the publisher previously created the TS_CHICAGO_DAILY tablespace.)

The following example creates a change table on the staging database that captures changes made to a source table in the source database. The example uses the sample table sh.products.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE(
       owner             => 'cdcpub',
       change_table_name => 'products_ct',
       change_set_name   => 'CHICAGO_DAILY',
       source_schema     => 'SH',
       source_table      => 'PRODUCTS',
       column_type_list  => 'PROD_ID NUMBER(6), PROD_NAME VARCHAR2(50),
          PROD_LIST_PRICE NUMBER(8,2)',
          JOB_ID VARCHAR2(10), DEPARTMENT_ID NUMBER(4)',
       capture_values    => 'both',
       rs_id             => 'y',
       row_id            => 'n',
       user_id           => 'n',
       timestamp         => 'n',
       object_id         => 'n',
       source_colmap     => 'n',
       target_colmap     => 'y',
       options_string    => 'TABLESPACE TS_CHICAGO_DAILY');
END;
/
```

This example creates a change table named products_ct within change set CHICAGO_DAILY. The column_type_list parameter identifies the columns captured by the change table. The source_schema and source_table parameters identify the schema and source table that reside in the source database, not the staging database.

The capture_values setting in the example indicates that for update operations, the change data will contain two separate rows for each row that changed: one row will contain the row values before the update occurred and the other row will contain the row values after the update occurred.

See "Managing Change Tables" on page 17-54 for more information.

Step 12 Staging Database Publisher: Enable the change set.

Because asynchronous change sets are always disabled when they are created, the publisher must alter the change set to enable it. The Oracle Streams capture and apply processes are started when the change set is enabled.

```
BEGIN
   DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
       change_set_name => 'CHICAGO_DAILY',
       enable_capture  => 'y');
END;
/
```

Step 13 Source Database DBA: Switch the redo log files at the source database.

To begin capturing data, a log file must be archived. The source database DBA can initiate the process by switching the current redo log file:

```
ALTER SYSTEM SWITCH LOGFILE;
```

Step 14 Staging Database Publisher: Grant access to subscribers.

The publisher controls subscriber access to change data by granting and revoking the SQL SELECT privilege on change tables for users and roles on the staging database. The publisher grants access to specific change tables. Without this step, a subscriber cannot access any change data. This example assumes that user subscriber1 already exists.

```
GRANT SELECT ON cdcpub.products_ct TO subscriber1;
```

The Change Data Capture asynchronous AutoLog system is now ready for subscriber1 to create subscriptions.

Subscribing to Change Data

When a publisher creates a change table, Change Data Capture assigns it a publication ID and maintains a list of all the publication IDs in the DBA_PUBLISHED_COLUMNS view. A **publication ID** is a numeric value that Change Data Capture assigns to each change table defined by the publisher.

The subscribers register their interest in one or more source tables, and obtain subscriptions to these tables. Assuming sufficient access privileges, the subscribers may subscribe to any source tables for which the publisher has created one or more change tables by doing one of the following:

- Specifying the source tables and columns of interest.

  When there are multiple publications that contain the columns of interest, then Change Data Capture selects one on behalf of the user.

- Specifying the publication IDs and columns of interest.

  When there are multiple publications on a single source table and these publications share some columns, the subscriber should specify publication IDs (rather than source tables) if any of the shared columns will be used in a single subscription.

The following steps provide an example to demonstrate both scenarios:

Step 1 Find the source tables for which the subscriber has access privileges.

The subscriber queries the DBA_SOURCE_TABLES view to see all the published source tables for which the subscriber has access privileges:

```
SELECT * FROM DBA_SOURCE_TABLES;

SOURCE_SCHEMA_NAME             SOURCE_TABLE_NAME
------------------------------ ------------------------------
SH                             PRODUCTS
```

Step 2 Find the change set names and columns for which the subscriber has access privileges.

The subscriber queries the DBA_PUBLISHED_COLUMNS view to see all the change sets, columns, and publication IDs for the sh.products table for which the subscriber has access privileges:

```
SELECT UNIQUE CHANGE_SET_NAME, COLUMN_NAME, PUB_ID
FROM DBA_PUBLISHED_COLUMNS
WHERE SOURCE_SCHEMA_NAME ='SH' AND SOURCE_TABLE_NAME = 'PRODUCTS';

CHANGE_SET_NAME     COLUMN_NAME        PUB_ID
----------------    ------------------ ------------
CHICAGO_DAILY       PROD_ID            41494
CHICAGO_DAILY       PROD_LIST_PRICE    41494
CHICAGO_DAILY       PROD_NAME          41494
```

Step 3 Create a subscription.

The subscriber calls the DBMS_CDC_SUBSCRIBE.CREATE_SUBSCRIPTION procedure to create a subscription.

The following example shows how the subscriber identifies the change set of interest (CHICAGO_DAILY), and then specifies a unique subscription name that will be used throughout the life of the subscription:

```
BEGIN
       DBMS_CDC_SUBSCRIBE.CREATE_SUBSCRIPTION(
       change_set_name   => 'CHICAGO_DAILY',
       description       => 'Change data for PRODUCTS',
       subscription_name => 'SALES_SUB');
END;
/
```

Step 4 Subscribe to a source table and the columns in the source table.

The subscriber calls the DBMS_CDC_SUBSCRIBE.SUBSCRIBE procedure to specify which columns of the source tables are of interest to the subscriber.

A subscription can contain one or more source tables referenced by the same change set.

In the following example, the subscriber wants to see the PROD_ID, PROD_NAME, and PROD_LIST_PRICE columns from the PRODUCTS table. Because all these columns are

contained in the same publication (and the subscriber has privileges to access that publication) as shown in the query in Step 2, the following call can be used:

```
BEGIN
       DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
       subscription_name => 'SALES_SUB',
       source_schema     => 'SH',
       source_table      => 'PRODUCTS',
       column_list       => 'PROD_ID, PROD_NAME, PROD_LIST_PRICE',
       subscriber_view   => 'SALES_VIEW');
END;
/
```

However, assume that for security reasons the publisher has not created a single change table that includes all these columns. Suppose that instead of the results shown in Step 2, the query of the DBA_PUBLISHED_COLUMNS view shows that the columns of interest are included in multiple publications as shown in the following example:

```
SELECT UNIQUE CHANGE_SET_NAME, COLUMN_NAME, PUB_ID
FROM DBA_PUBLISHED_COLUMNS
WHERE SOURCE_SCHEMA_NAME ='SH' AND SOURCE_TABLE_NAME = 'PRODUCTS';

CHANGE_SET_NAME     COLUMN_NAME        PUB_ID
----------------    ------------------ ------------
CHICAGO_DAILY       PROD_ID            34883
CHICAGO_DAILY       PROD_NAME          34885
CHICAGO_DAILY       PROD_LIST_PRICE    34883
CHICAGO_DAILY       PROD_ID            34885
```

This returned data shows that the PROD_ID column is included in both publication 34883 and publication 34885. A single subscribe call must specify columns available in a single publication. Therefore, if the subscriber wants to subscribe to columns in both publications, using PROD_ID to join across the subscriber views, then the subscriber must use two calls, each specifying a different publication ID:

```
BEGIN
       DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
       subscription_name => 'MULTI_PUB',
       publication_id    => 34885,
       column_list       => 'PROD_ID, PROD_NAME',
       subscriber_view   => 'prod_idname');

       DBMS_CDC_SUBSCRIBE.SUBSCRIBE(
       subscription_name => 'MULTI_PUB',
       publication_id    => 34883,
       column_list       => 'PROD_ID, PROD_LIST_PRICE',
       subscriber_view   => 'prod_price');
END;
/
```

Note that each DBMS_CDC_SUBSCRIBE.SUBSCRIBE call specifies a unique subscriber view.

Step 5 Activate the subscription.

The subscriber calls the DBMS_CDC_SUBSCRIBE.ACTIVATE_SUBSCRIPTION procedure to activate the subscription.

A subscriber calls this procedure when finished subscribing to source tables (or publications), and ready to receive change data. Whether subscribing to one or multiple source tables, the subscriber needs to call the ACTIVATE_SUBSCRIPTION procedure only once.

The ACTIVATE_SUBSCRIPTION procedure creates empty subscriber views. At this point, DBMS_CDC_SUBSCRIBE.SUBSCRIBE calls can be made.

```
BEGIN
   DBMS_CDC_SUBSCRIBE.ACTIVATE_SUBSCRIPTION(
       subscription_name => 'SALES_SUB');
END;
/
```

Step 6 Get the next set of change data.

The subscriber calls the DBMS_CDC_SUBSCRIBE.EXTEND_WINDOW procedure to get the next available set of change data. This sets the high boundary of the subscription window. For example:

```
 BEGIN
   DBMS_CDC_SUBSCRIBE.EXTEND_WINDOW(
       subscription_name => 'SALES_SUB');
END;
/
```

If this is the subscriber's first call to the EXTEND_WINDOW procedure, then the subscription window contains all the change data in the publication. Otherwise, all the new change data that was created since the last call to the EXTEND_WINDOW procedure is added to the subscription window.

If no new change data has been added, then the subscription window remains unchanged.

Step 7 Read and query the contents of the subscriber views.

The subscriber uses the SQL SELECT statement on the subscriber view to query the change data (within the current boundaries of the subscription window). The subscriber can do this for each subscriber view in the subscription. For example:

```
SELECT PROD_ID, PROD_NAME, PROD_LIST_PRICE FROM SALES_VIEW;

PROD_ID    PROD_NAME                                          PROD_LIST_PRICE
---------- -------------------------------------------------- ---------------
30         And 2 Crosscourt Tee Kids                          14.99
30         And 2 Crosscourt Tee Kids                          17.66
10         Gurfield& Murks Pleated Trousers                   17.99
10         Gurfield& Murks Pleated Trousers                   21.99
```

The subscriber view name, SALES_VIEW, was specified when the subscriber called the DBMS_CDC_SUBSCRIBE.SUBSCRIBE procedure in Step 4.

Step 8 Indicate that the current set of change data is no longer needed.

The subscriber uses the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure to let Change Data Capture know that the subscriber no longer needs the current set of change data. This helps Change Data Capture to manage the amount of data in the change table and sets the low boundary of the subscription window. Calling the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure causes the subscription window to be empty. See "Purging Change Tables of Unneeded Data" on page 17-61 for details on how the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW works.

For example:

```
BEGIN
   DBMS_CDC_SUBSCRIBE.PURGE_WINDOW(
       subscription_name => 'SALES_SUB');
```

```
END;
/
```

> **Note:** Before using `DBMS_CDC_PUBLISH.PURGE`, call `DBMS_CDC_SUBSCRIBE.EXTEND_WINDOW`, which protects the data that is still needed by the subscriber and makes the publisher aware that there is some important data that should not be purged.

Step 9 Repeat Steps 6 through 8.

The subscriber repeats Steps 6 through 8 as long as the subscriber is interested in additional change data.

Step 10 End the subscription.

The subscriber uses the `DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION` procedure to end the subscription. This is necessary to prevent the publications that underlie the subscription from holding change data indefinitely.

```
BEGIN
   DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION(
       subscription_name => 'SALES_SUB');
END;
/
```

Managing Published Data

This section provides information about the management tasks involved in managing change sets and change tables. For the most part, these tasks are the responsibility of the publisher. However, to purge unneeded data from the change tables, both the publisher and the subscribers have responsibilities as described in "Purging Change Tables of Unneeded Data" on page 17-61

The following topics are covered in this section:

- Managing Asynchronous Change Sources
- Managing Asynchronous Change Sets
- Managing Change Tables
- Exporting and Importing Change Data Capture Objects Using Oracle Data Pump
- Impact on Subscriptions When the Publisher Makes Changes

Managing Asynchronous Change Sources

This section provides information about tasks that the publisher can perform to manage asynchronous change sources. The following topic is covered:

- Enabling And Disabling Asynchronous Distributed HotLog Change Sources

Enabling And Disabling Asynchronous Distributed HotLog Change Sources

The publisher can enable and disable asynchronous Distributed HotLog change sources. When a change source is disabled, it cannot process new change data until the change source is enabled.

Asynchronous Distributed HotLog change sources are always created disabled.

The publisher can enable the PRODUCTS_SOURCE asynchronous Distributed HotLog change source with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_HOTLOG_CHANGE_SOURCE(
change_source_name => 'PRODUCTS_SOURCE',
enable_source      => 'y');
END;
/
```

The Oracle Streams capture process for the change source is started when the change source is enabled.

The publisher can disable the PRODUCTS_SOURCE asynchronous Distributed HotLog change source using the same call, but with enable_source set to 'n'. The Oracle Streams capture process for the change source is stopped when the change source is disabled.

An asynchronous Distributed HotLog change source and its change sets must be enabled separately in order to process change data and populate change tables. See "Enabling and Disabling Asynchronous Change Sets" on page 17-50 for more information.

Although a disabled change source cannot process new change data, it does not lose any change data, provided the necessary archived redo log files remain available until the change source and its changes sets are enabled and can process them. Oracle recommends that change sources and change sets be enabled as much as possible to avoid accumulating archived redo log files. See "Asynchronous Change Data Capture and Redo Log Files" on page 17-68 for more information.

Managing Asynchronous Change Sets

This section provides information about tasks that the publisher can perform to manage asynchronous change sets. The following topics are covered:

- Creating Asynchronous Change Sets with Starting and Ending Dates
- Enabling and Disabling Asynchronous Change Sets
- Stopping Capture on DDL for Asynchronous Change Sets
- Recovering from Errors Returned on Asynchronous Change Sets

Creating Asynchronous Change Sets with Starting and Ending Dates

Change sets associated with asynchronous HotLog and AutoLog change sources can optionally specify starting and ending dates to limit the change data they capture. (Change sets associated with Distributed HotLog change sources cannot specify starting and ending dates). A change set with no starting date begins capture with the earliest available change data. A change set with no ending date continues capturing change data indefinitely.

The following example creates a change set, PRODUCTS_SET, in the AutoLog change source, HQ_SOURCE, that starts capture two days from now and continues indefinitely:

```
BEGIN
DBMS_CDC_PUBLISH.CREATE_CHANGE_SET(
change_set_name    => 'PRODUCTS_SET',
description        => 'Products Application Change Set',
change_source_name => 'HQ_SOURCE',
stop_on_ddl        => 'Y',
```

```
begin_date         => sysdate+2);
END;
/
```

Enabling and Disabling Asynchronous Change Sets

The publisher can enable and disable asynchronous change sets. When a change set is disabled, it cannot process new change data until the change set is enabled.

Synchronous change sets are always created enabled. Asynchronous change sets are always created disabled.

The publisher can enable the `PRODUCTS_SET` asynchronous change set with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
change_set_name => 'PRODUCTS_SET',
enable_capture  => 'y');
END;
/
```

For asynchronous HotLog and AutoLog change sets, the Oracle Streams capture and apply processes are started when the change set is enabled. For asynchronous Distributed HotLog change sets, the Oracle Streams apply process is started when the change set is enabled.

The publisher can disable the `PRODUCTS_SET` asynchronous change set using the same call, but with `enable_capture` set to `'n'`. For asynchronous HotLog and AutoLog change sets, the Oracle Streams capture and apply processes are stopped when the change set is disabled. For asynchronous Distributed HotLog change sets, the Oracle Streams apply process is stopped when the change set is disabled.

An asynchronous Distributed HotLog change source and its change sets must be enabled separately in order to process change data and populate change tables. See "Enabling And Disabling Asynchronous Distributed HotLog Change Sources" on page 17-48 for more information.

Although a disabled change set cannot process new change data, it does not lose any change data if the necessary archived redo log files remain available until the change set is enabled and processes them. Oracle recommends that change sets be enabled as much as possible to avoid accumulating archived redo log files. See "Asynchronous Change Data Capture and Redo Log Files" on page 17-68 for more information.

Change Data Capture can automatically disable an asynchronous change set if there is an internal capture error. The publisher must check the alert log for more information, take any necessary actions to adjust to the DDL or recover from the internal error, and explicitly enable the change set. See "Recovering from Errors Returned on Asynchronous Change Sets" on page 17-51 for more information.

Stopping Capture on DDL for Asynchronous Change Sets

The publisher can specify that a change set be automatically disabled by Change Data Capture if DDL is encountered. Some DDL commands can adversely affect capture, such as dropping a source table column that is being captured. If the change set stops on DDL, the publisher has a chance to analyze and fix the problem before capture proceeds. If the change set does not stop on DDL, internal capture errors are possible after DDL occurs.

The publisher can specify whether a change set stops on DDL when creating or altering the change set. The publisher can alter the PRODUCTS_SET change set to stop on DDL with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
change_set_name  => 'PRODUCTS_SET',
stop_on_ddl      => 'y');
END;
/
```

The publisher can alter the PRODUCTS_SET change set so that it does not stop on DDL by setting stop_on_ddl to 'n'.

If a DDL statement causes processing to stop, a message is written to the alert log indicating the DDL statement and change set involved. For example, if a TRUNCATE TABLE DDL statement causes the PRODUCTS_SET change set to stop processing, the alert log contains lines such as the following:

```
Change Data Capture received DDL for change set PRODUCTS_SET
Change Data Capture received DDL and stopping: truncate table products
```

Because they do not affect the column data itself, the following DDL statements do not cause Change Data Capture to stop capturing change data when the stop_on_ddl parameter is set to 'Y':

- ANALYZE TABLE
- LOCK TABLE
- GRANT privileges to access a table
- REVOKE privileges to access a table
- COMMENT on a table
- COMMENT on a column

These statements can be issued on the source database without concern for their impact on Change Data Capture processing. For example, when an ANALYZE TABLE command is issued on the PRODUCTS source table, the alert log on the staging database will contain a line similar to the following when the stop_on_ddl parameter is set to 'Y':

```
Change Data Capture received DDL and ignoring: analyze table products compute statistics
```

Recovering from Errors Returned on Asynchronous Change Sets

Errors during asynchronous Change Data Capture are possible due to a variety of circumstances. If a change set stops on DDL, that DDL must be removed before capture can continue. If a change set does not stop on DDL, but a DDL change occurs that affects capture, it can result in an Oracle error. There are also system conditions that can cause Oracle errors, such as being out of disk space.

In all these cases, the change set is disabled and marked as having an error. Subscriber procedures detect when a change set has an error and return the following message:

```
ORA-31514: change set disabled due to capture error
```

The publisher must check the alert log for more information and attempt to fix the underlying problem. The publisher can then attempt to recover from the error by calling ALTER_CHANGE_SET with the recover_after_error and remove_ddl parameters set to 'y'.

The publisher can repeat this procedure as many times as necessary to resolve the problem. When recovery succeeds, the error is removed from the change set and the publisher can enable the asynchronous change set (as described in "Enabling and Disabling Asynchronous Change Sets" on page 17-50).

The publisher should be aware that if there are multiple consecutive DDLs, the change set stops for each one separately. For example, suppose there are two consecutive DDLs. When the change set stops for the first DDL, the publisher must remove that DDL and then re-enable capture for the change set. The change set will then stop for the second DDL. The publisher must remove the second DDL and re-enable capture for the change set again.

If more information is needed to resolve capture errors, the publisher can query the DBA_APPLY_ERROR view to see information about Streams apply errors; capture errors correspond to Streams apply errors. The publisher must always use the DBMS_CDC_PUBLISH.ALTER_CHANGE_SET procedure to recover from capture errors because both Streams and Change Data Capture actions are needed for recovery and only the DBMS_CDC_PUBLISH.ALTER_CHANGE_SET procedure performs both sets of actions. See *Oracle Streams Concepts and Administration* for information about the error queue and apply errors.

The following two scenarios demonstrate how a publisher might investigate and then recover from two different types of errors returned to Change Data Capture:

An Error Due to Running Out of Disk Space The publisher can view the contents of the alert log to determine which error is being returned for a given change set and which SCN is not being processed. For example, the alert log may contain lines such as the following (where LCR refers to a logical change record):

```
Change Data Capture has encountered error number: 1688 for change set:
   CHICAGO_DAILY
Change Data Capture did not process LCR with scn 219337
```

The publisher can determine the message associated with the error number specified in the alert log by querying the DBA_APPLY_ERROR view for the error message text, where the APPLY_NAME in the DBA_APPLY_ERROR view equals the APPLY_NAME of the change set specified in the alert log. For example:

```
SQL> SELECT ERROR_MESSAGE FROM DBA_APPLY_ERROR
       WHERE APPLY_NAME =
      (SELECT APPLY_NAME FROM ALL_CHANGE_SETS WHERE SET_NAME ='CHICAGO_DAILY');

ERROR_MESSAGE
--------------------------------------------------------------------------------
ORA-01688: unable to extend table LOGADMIN.CT1 partition P1 by 32 in tablespace
    TS_CHICAGO_DAILY
```

After taking action to fix the problem that is causing the error, the publisher can attempt to recover from the error. For example, the publisher can attempt to recover the CHICAGO_DAILY change set after an error with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
  change_set_name     => 'CHICAGO_DAILY',
  recover_after_error => 'y');
END;
/
```

If the recovery does not succeed, then an error is returned and the publisher can take further action to attempt to resolve the problem. The publisher can retry the recovery procedure as many times as necessary to resolve the problem.

> **Note:** When recovery succeeds, the publisher must remember to enable the change set. After being enabled, the change data capture operation will proceed with the logical change record (LCR) where the error occurred. No change data will be lost.

An Error Due to Stopping on DDL Suppose a SQL `TRUNCATE TABLE` statement is issued against the `PRODUCTS` source table and the `stop_on_ddl` parameter is set to `'Y'`, then an error such as the following is returned from an attempt to enable the change set:

```
ERROR at line 1:
ORA-31468: cannot process DDL change record
ORA-06512: at "SYS.DBMS_CDC_PUBLISH", line 79
ORA-06512: at line 2
```

The alert log will contain lines similar to the following:

```
Mon Jun  9 16:13:44 2003
Change Data Capture received DDL for change set PRODUCTS_SET
Change Data Capture received DDL and stopping: truncate table products
Mon Jun  9 16:13:50 2003
Change Data Capture did not process LCR with scn 219777
Streams Apply Server P001 pid=19 OS id=11730 stopped
Streams Apply Reader P000 pid=17 OS id=11726 stopped
Streams Apply Server P000 pid=17 OS id=11726 stopped
Streams Apply Server P001 pid=19 OS id=11730 stopped
Streams AP01 with pid=15, OS id=11722 stopped
```

Because the `TRUNCATE TABLE` statement removes all rows from a table, the publisher will want to notify subscribers before taking action to re-enable Change Data Capture processing. He or she might suggest to subscribers that they purge and extend their subscription windows. The publisher can then attempt to restore Change Data Capture processing by altering the change set and specifying the `remove_ddl => 'Y'` parameter along with the `recover_after_error => 'Y'` parameter, as follows:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
  change_set_name     => 'PRODUCTS_SET',
  recover_after_error => 'y',
  remove_ddl          => 'y');
END;
/
```

After this procedure completes, the alert log will contain lines similar to the following:

```
Mon Jun  9 16:20:17 2003
Change Data Capture received DDL and ignoring: truncate table products
The scn for the truncate statement is 202998
```

Now, the publisher must enable the change set. All change data that occurred after the `TRUNCATE TABLE` statement will be reflected in the change table. No change data will be lost.

Managing Synchronous Change Sets

The publisher can enable and disable synchronous change sets. When a change set is disabled, it cannot process new change data until the change set is enabled.

Enabling and Disabling Synchronous Change Sets

Synchronous change sets are always created enabled. Asynchronous change sets are always created disabled.

The publisher can enable the `PRODUCTS_SET` synchronous change set with the following call:

```
BEGIN
DBMS_CDC_PUBLISH.ALTER_CHANGE_SET(
change_set_name => 'PRODUCTS_SET',
enable_capture  => 'y');
END;
/
```

The publisher can disable the `PRODUCTS_SET` synchronous change set using the same call, but with `enable_capture` set to `'n'`.

Managing Change Tables

All change table management tasks are the responsibility of the publisher with one exception: purging change tables of unneeded data. This task requires action from both the publisher and the subscriber to work most effectively.

The following topics are discussed in this section:

- Creating Change Tables
- Understanding Change Table Control Columns
- Understanding TARGET_COLMAP$ and SOURCE_COLMAP$ Values
- Using Change Markers
- Controlling Subscriber Access to Change Tables
- Purging Change Tables of Unneeded Data
- Dropping Change Tables

Creating Change Tables

When creating change tables, the publisher should be aware that Oracle recommends the following:

- For all modes of Change Data Capture, publishers should not create change tables in system tablespaces.

  Either of the following methods can be used to ensure that change tables are created in tablespaces managed by the publisher. The first method creates all the change tables created by the publisher in a single tablespace, while the second method allows the publisher to specify a different tablespace for each change table.

  – When the database administrator creates the account for the publisher, he or she can specify a default tablespace. For example:

    ```
    CREATE USER cdcpub DEFAULT TABLESPACE ts_cdcpub;
    ```

– When the publisher creates a change table, he or she can use the `options_string` parameter to specify a tablespace for the change table being created. See Step 4 in "Performing Synchronous Publishing" on page 17-26 for an example.

If both methods are used, the tablespace specified by the publisher in the `options_string` parameter takes precedence over the default tablespace specified in the SQL `CREATE USER` statement.

- For asynchronous Change Data Capture, the publisher should be certain that the source table that will be referenced in a `DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE` procedure has been created prior to calling this procedure, particularly if the change set that will be specified in the procedure has the `stop_on_ddl` parameter set to `'Y'`.

  Suppose the publisher created a change set with the `stop_on_ddl` parameter set to `'Y'`, then created the change table, and then the source table was created. In this scenario, the DDL that creates the source table would trigger the `stop_on_ddl` condition and cause Change Data Capture processing to stop.

  Note: The publisher must not attempt to control a change table's partitioning properties. Change Data Capture automatically manages the change table partitioning as part of its change table management.

- For asynchronous Change Data Capture, the source database DBA should create an unconditional log group for all source table columns that will be captured in a change table. This should be done before any change tables are created on a source table. If an unconditional log group is not created for source table columns to be captured, then when an update DML operation occurs, some unchanged user column values in change tables will be null instead of reflecting the actual source table value. This will require the publisher to evaluate the `TARGET_COLMAP$` control column to distinguish unchanged column values from column values that are actually null. See "Asynchronous Change Data Capture and Supplemental Logging" on page 17-70 for information on creating unconditional log groups and see "Understanding Change Table Control Columns" on page 17-55 for information on control columns.

Understanding Change Table Control Columns

A change table consists of two things: the change data itself, which is stored in a database table, and the system metadata necessary to maintain the change table, which includes control columns.

Table 17–11 describes the control columns for a change table, including the column name, datatype, mode, whether the column is optional or not, and a description.

The mode indicates the type of Change Data Capture associated with the column. A value of `ALL` indicates that the column is associated with the synchronous mode and all modes of asynchronous Change Data Capture. Note that for both synchronous and asynchronous Change Data Capture, if the subscriber wants a query of a subscriber view to return DML changes in the order in which they occurred, the query should order data by `CSCN$` and then `RSID$`.

A control column is considered optional if the publisher can choose to exclude it from a change table. Table 17–11 identifies which control columns are optional under what modes. All optional control columns are specified with a parameter in the `DBMS_CDC_`

PUBLISH.CREATE_CHANGE_TABLE procedure. The syntax for the DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE procedure is documented in *Oracle Database PL/SQL Packages and Types Reference*.

Table 17–11 Control Columns for a Change Table

Column	Datatype	Mode	Optional Column	Description
OPERATION$	CHAR(2)	All	No	The value in this column can be any one of the following[1]: I: Indicates this row represents an insert operation UO: Indicates this row represents the before-image of an updated source table row for the following cases: ■ Asynchronous Change Data Capture ■ Synchronous Change Data Capture when the change table includes a primary key-based object ID and a captured column that is not a primary key has changed. UU: Indicates this row represents the before-image of an updated source table row for synchronous Change Data Capture, in cases other than those represented by UO. UN: Indicates this row represents the after-image of an updated source table row. D: Indicates this row represents a delete operation.
CSCN$	NUMBER	All	No	For synchronous CDC, the value for the EXTEND_WINDOW operation. For asynchronous CDC, the value for the DML operation.
DDLDESC$	CLOB	Asynchronous	Yes	A clob containing the actual DDL statement executed.
DDLPDOBJN$	NUMBER	Asynchronous	Yes	This is not used in this release.
DDLOPER$	NUMBER	Asynchronous	Yes	A bit vector that indicates what kind of DDL operation happened.
RSID$	NUMBER	All	Yes	Unique row sequence ID within this transaction.[2] The RSID$ column reflects an operation's capture order within a transaction, but not across transactions. The publisher cannot use the RSID$ column value by itself to order committed operations across transactions; it must be used in conjunction with the CSCN$ column value.
SOURCE_COLMAP$	RAW(128)	Synchronous	Yes	Bit mask[3] of updated columns in the source table.
TARGET_COLMAP$	RAW(128)	All	Yes	Bit mask[3] of updated columns in the change table.
COMMIT_TIMESTAMP$	DATE	All	No	Commit time of this transaction.
TIMESTAMP$	DATE	All	Yes	Time when the operation occurred in the source table.

Table 17–11 (Cont.) Control Columns for a Change Table

Column	Datatype	Mode	Optional Column	Description
USERNAME$	VARCHAR2(30)	All	Yes	Name of the user who caused the operation. Value is always NULL for change sources created on Oracle Databases prior to release 10.2.
ROW_ID$	ROW_ID	All	Yes	Row ID of affected row in source table. Value is always NULL for change sources created on Oracle Databases prior to release 10.2.
XIDUSN$	NUMBER	Asynchronous	No	Transaction ID undo segment number.
XIDSLT$	NUMBER	Asynchronous	No	Transaction ID slot number.
XIDSEQ$	NUMBER	Asynchronous	No	Transaction ID sequence number.
SYS_NC_OID$	RAW(16)	Synchronous	Yes	Object ID.

[1] If you specify a query based on this column, specify the I or D column values as "I " or "D ", respectively. The OPERATION$ column is a 2-character column; values are left-justified and space-filled. A query that specifies a value of "I" or "D" will return no values.

[2] You can use the RSID$ column to associate the after-image with the before-image of a given operation. The value of the after-image RSID$ column always matches the value of the before-image RSID$ column value for a given update operation.

[3] A bit mask is an array of binary values that indicate which columns in a row have been updated.

Understanding TARGET_COLMAP$ and SOURCE_COLMAP$ Values

The TARGET_COLMAP$ and SOURCE_COLMAP$ columns are used to indicate which columns in a row have changed. The TARGET_COLMAP$ column indicates which columns in the change table row have changed. The SOURCE_COLMAP$ column (which is included for synchronous change tables only) indicates which columns in a source table row have changed.

Because the datatype of the TARGET_COLMAP$ and the SOURCE_COLMAP$ columns is RAW(128), each column can hold 128 bytes of binary information. This binary information consists of words strung together with the low order byte on the left and the high order byte on the right. By default, when the field is displayed, the system interprets each byte and displays a value for each. Example 17–1 provides a sample TARGET_COLMAP$ value. The text following the example explains how to interpret this data.

Example 17–1 Sample TARGET_COLMAP$ VALUE

```
FE1100000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000
```

In Example 17–1, the first 'FE' is the low order byte and the last '00' is the high order byte. To correctly interpret the meaning of the values, you must consider which bits are set in each byte. The bits in the bitmap are counted starting at zero. The first bit is bit 0, the second bit is bit 1, and so on. Bit 0 is always ignored. For the other bits, if a particular bit is set to 1, it means that the value for that column has been changed.

To interpret the string of bytes as presented in the Example 17–1, you read from left to right. The first byte is the string 'FE'. Broken down into bits (again from left to right) this string is "1111 1110", which maps to columns " 7,6,5,4 3,2,1,-" in the change table (where the hyphen represents the ignored bit). The first bit tells you if column 7 in the change table has changed. The right-most bit is ignored. The values in Example 17–1 indicate that the first 7 columns have a value present. This is typical - the first several columns in a change table are control columns.

The next byte in Example 17–1 is the string '11'. Broken down into bits, this string is "0001 0001", which maps to columns "15,14,13,12 11,10,9,8" in the change table. These bits indicate that columns 8 and 12 are changed. Columns 9, 10, 11, 13, 14, 15, are not changed. The rest of the string is all '00', indicating that none of the other columns has been changed.

A publisher can issue the following query to determine the mapping of column numbers to column names:

```
SELECT COLUMN_NAME, COLUMN_ID FROM ALL_TAB_COLUMNS
WHERE OWNER='PUBLISHER_STEWART' AND TABLE_NAME='MY_CT';

COLUMN_NAME                    COLUMN_ID
------------------------------ ----------
OPERATION$                              1
CSCN$                                   2
COMMIT_TIMESTAMP$                       3
XIDUSN$                                 4
XIDSLT$                                 5
XIDSEQ$                                 6
RSID$                                   7
TARGET_COLMAP$                          8
C_ID                                    9
C_KEY                                  10
C_ZIP                                  11

COLUMN_NAME                     COLUMN_ID
------------------------------ ----------
C_DATE                                 12
C_1                                    13
C_3                                    14
C_5                                    15
C_7                                    16
C_9                                    17
```

Using Example 17–1, the publisher can conclude that following columns were changed in the particular change row in the change table represented by this TARGET_COLMAP$ value: OPERATION$, CSCN$, COMMIT_TIMESTAMP$, XIDUSN$, XIDSLT$, XIDSEQ$, RSID$, TARGET_COLMAP$, and C_DATE.

Note that Change Data Capture generates values for all control columns in all change rows, so the bits corresponding to control columns are always set to 1 in every TARGET_COLMAP$ column. Bits that correspond to user columns that have changed are set to 1 for the OPERATION$ column values UN and I, as appropriate. (See Table 17–11 for information about the OPERATION$ column values.) Also note that if you use asynchronous CDC, you will always see a 1 for the columns that were included in the UPDATE SET clause, irrespective of whether the value was actually changed from its original value. For synchronous CDC, you will only see a 1 if the value was changed from its original value.

A common use for the values in the TARGET_COLMAP$ column is for determining the meaning of a null value in a change table. A column value in a change table can be null for two reasons: the value was changed to null by a user or application, or Change Data Capture inserted a null value into the column because a value was not present in the redo data from the source table. If a user changed the value to null, the bit for that column will be set to 1; if Change Data Capture set the value to null, then the column will be set to 0.

Values in the SOURCE_COLMAP$ column are interpreted in a similar manner, with the following exceptions:

- The SOURCE_COLMAP$ column refers to columns of source tables, not columns of change tables.
- The SOURCE_COLMAP$ column does not reference control columns because these columns are not present in the source table.
- Changed source columns are set to 1 in the SOURCE_COLMAP$ column for OPERATION$ column values UO, UU, UN, and I, as appropriate. (See Table 17–11 for information about the OPERATION$ column values.)
- The SOURCE_COLMAP$ column is valid only for synchronous change tables.

Using Change Markers

The following example illustrates how to use change markers. First, create the change set:

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_SET (
      CHANGE_SET_NAME     => 'async_set1',
      CHANGE_SOURCE_NAME  => 'HOTLOG_SOURCE',
      STOP_ON_DDL         => 'N',
      BEGIN_DATE          => SYSDATE,
      END_DATE            => SYSDATE+5);

PL/SQL procedure successfully completed
```

Next, create the change table. Note that this should have the three DDL markers ddloper$, ddldesc$, and ddlpdobjn$.

```
BEGIN
   DBMS_CDC_PUBLISH.CREATE_CHANGE_TABLE (
      OWNER             => 'tcdcpub',
      CHANGE_TABLE_NAME => 'cdc_psales_act',
      CHANGE_SET_NAME   => 'async_set1',
      SOURCE_SCHEMA     => 'tcdcpub',
      SOURCE_TABLE      => 'cdc_psales',
      COLUMN_TYPE_LIST  => 'PROD_IC NUMBER(6)
         NUMBER, TIME_ID DATE',
      CAPTURE_VALUES    => 'both',
      RS_ID             => 'N',
      ROW_ID            => 'N',
      USER_ID           => 'N',
      TIMESTAMP         => 'N',
      OBJECT_ID         => 'N',
      SOURCE_COLMAP     => 'N',
      TARGET_COLMAP     => 'Y',
      OPTIONS_STRING    => NULL,
      DDL_MARKERS       => 'Y');
END;
/

PL/SQL procedure successfully completed.
describe cdc_psales_act;

Name                    Null?     Type
----------------------------------------
OPERATION$                        CHAR(2)
CSCN$                             NUMBER
COMMIT_TIMESTAMP                  DATE
XIDUSN$                           NUMBER
XIDSLT$                           NUMBER
```

```
XIDSEQ$                         NUMBER
TARGET_COLMAP$                  RAW(128)
DDLOPER$                        NUMBER
DDLDESC$                        CLOB
DDLPDOBJN$                      NUMBER
PROD_D                          NUMBER(6)
CUST_ID                         NUMBER
TIME_ID                         DATE
```

Then, enable capture for the change set:

```
BEGIN
   DBMS_CDC_PUBLISH.ALTER_CHANGE_SET (
      CHANGE_SET_NAME  => 'asynch_set1',
      ENABLE_CAPTURE   => 'Y');
END;
.
PL/SQL procedure successfully completed
```

Finally, issue a DDL statement and see if it is captured:

```
ALTER TABLE cdc_psales DROP PARTITION Dec_06;

SELECT ddloper$, DECODE(ddloper$, NULL, 'NULL',
       DBMS_CDC_PUBLISH.GET_DDLOPER(ddloper$))
          AS DDL_OPER
FROM cdc_psales_act
WHERE DDLOPER$ IS NOT NULL
ORDER BY cscn$;

ddloper$            DDL_OPER
---------------------------------
512                 Drop Partition
1 row selected.

SELECT ddldesc$
FROM cdc_psales_act
WHERE ddloper$
IS NOT NULL
ORDER BY cscn;

DDLDESC$
------------------------
alter table cdc_psales drop partition Dec_06
1 row selected.
```

Controlling Subscriber Access to Change Tables

The publisher grants privileges to subscribers to allow them access to change tables. Because privileges on source tables are not propagated to change tables, a subscriber might have privileges to perform a `SELECT` operation on a source table, but might not have privileges to perform a `SELECT` operation on a change table that references that source table.

The publisher controls subscriber access to change data by using the SQL `GRANT` and `REVOKE` statements to grant and revoke the `SELECT` privilege on change tables for users and roles. The publisher must grant the `SELECT` privilege before a subscriber can subscribe to the change table.

The publisher must not grant any DML access (use of `INSERT`, `UPDATE`, or `DELETE` statements) to the subscribers on the change tables because a subscriber might

inadvertently change the data in the change table, making it inconsistent with its source. Furthermore, the publisher should avoid creating change tables in schemas to which subscribers have DML access.

Purging Change Tables of Unneeded Data

This section describes purge operations. For optimum results, purge operations require action from the subscribers. Each subscriber indicates when he or she is done using change data, and then Change Data Capture or the publisher actually removes (purges) data that is no longer being used by any subscriber from the change table, as follows:

- Subscriber

  When finished using change data, a subscriber must call the `DBMS_CDC_SUBSCRIBE.PURGE_WINDOW` procedure. This indicates to Change Data Capture and the publisher that the change data is no longer needed by this subscriber. The `DBMS_CDC_SUBSCRIBE.PURGE_WINDOW` procedure does not physically remove rows from the change tables; however, the data is removed logically from the subscriber view and can no longer be selected.

  In addition, as shown in "Subscribing to Change Data" beginning on page 17-44, the subscriber should call the `DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION` procedure to drop unneeded subscriptions.

  See *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_CDC_SUBSCRIBE.DROP_SUBSCRIPTION` and the `DBMS_CDC_SUBSCRIBE.PURGE_WINDOW` procedures.

- Change Data Capture

  Change Data Capture creates a purge job using the `DBMS_SCHEDULER` package (which runs under the account of the publisher who created the first change table). This purge job calls the `DBMS_CDC_PUBLISH.PURGE` procedure to remove data that subscribers are no longer using from the change tables. This job has the name `cdc$_default_purge_job`. By default, this job runs every 24 hours. You can change the schedule of this job using `DBMS_SCHEDULER.SET_ATTRIBUTE` and set the `repeat_interval` attribute. You can verify or modify any other attributes with the `DBMS_SCHEDULER` package.

  This ensures that the size of the change tables does not grow without limit. The call to the `DBMS_CDC_PUBLISH.PURGE` procedure evaluates all active subscription windows to determine which change data is still needed. It will not purge any data that could be referenced by one or more subscribers with active subscription windows.

  See *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SCHEDULER` package.

- Publisher

  The publisher can manually execute a purge operation at any time. The publisher has the ability to perform purge operations at a finer granularity than the automatic purge operation performed by Change Data Capture. There are three purge operations available to the publisher:

  – `DBMS_CDC_PUBLISH.PURGE`

    Purges all change tables on the staging database. This is the same `PURGE` operation as is performed automatically by Change Data Capture.

  – `DBMS_CDC_PUBLISH.PURGE_CHANGE_SET`

Purges all the change tables in a named change set.

– DBMS_CDC_PUBLISH.PURGE_CHANGE_TABLE

  Purges a named changed table.

Thus, calls to the DBMS_CDC_SUBSCRIBE.PURGE_WINDOW procedure by subscribers and calls to the PURGE procedure by Change Data Capture (or one of the PURGE procedures by the publisher) work together: when each subscriber purges a subscription window, it indicates change data that is no longer needed; the PURGE procedure evaluates the sum of the input from all the subscribers before actually purging data. You can purge a table by using the PURGE_CHANGE_TABLE procedure, and you can purge all the change tables in a change set with the PURGE_CHANGE_SET procedure.

If force is set to N, Oracle Database tries to purge with the most efficient means (using a table lock), but does not guarantee it. Setting force to Y guarantees this purge.

Note that it is possible that a subscriber could fail to call PURGE_WINDOW, with the result being that unneeded rows would not be deleted by the purge job. The publisher can query the DBA_SUBSCRIPTIONS view to determine if this is happening, however, the publisher should first consider that subscribers may still be using the change data. In extreme circumstances, a publisher may decide to manually drop an active subscription so that space can be reclaimed. One such circumstance is a subscriber that is an applications program that fails to call the PURGE_WINDOW procedure when appropriate. The DBMS_CDC_PUBLISH.DROP_SUBSCRIPTION procedure lets the publisher drop active subscriptions if circumstances require it; however, the publisher should first consider that subscribers may still be using the change data.

Before using DBMS_CDC_PUBLISH.PURGE, it is important to execute DBMS_CDC_SUBSCRIBE.EXTEND_WINDOW, which protects the data and ensures that the publisher is aware of the required data that is needed by the subscriber. Without running the EXTEND_WINDOW procedure, the purge job will work as expected and it will purge all the data irrespective of whether data has been consumed or not by the subscriber.

Dropping Change Tables

To drop a change table, the publisher must call the DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE procedure. This procedure ensures that both the change table itself and the Change Data Capture metadata for the table are dropped. If the publisher tries to use a SQL DROP TABLE statement on a change table, it will fail with the following error:

```
ORA-31496 must use DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE to drop change tables
```

The DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE procedure also safeguards the publisher from inadvertently dropping a change table while there are active subscribers using the change table. If DBMS_CDC_PUBLISH.DROP_CHANGE_TABLE is called while subscriptions are active, the procedure will fail with the following error:

```
ORA-31424 change table has active subscriptions
```

If the publisher still wants to drop the change table, in spite of active subscriptions, he or she must call the DROP_CHANGE_TABLE procedure using the force_flag => 'Y' parameter. This tells Change Data Capture to override its normal safeguards and allow the change table to be dropped despite active subscriptions. The subscriptions will no longer be valid, and subscribers will lose access to the change data.

Note: The DROP USER CASCADE statement will drop all the publisher's change tables, and if any other users have active subscriptions to the (dropped) change table, these will no longer be valid. In addition to dropping the change tables, the DROP USER CASCADE statement drops any change sources, change sets, and subscriptions that are owned by the user specified in the DROP USER CASCADE statement.

Exporting and Importing Change Data Capture Objects Using Oracle Data Pump

Oracle Data Pump is the supported export and import utility for Change Data Capture.

The following sections describe restrictions, provide examples, and describe publisher considerations for using Oracle Data Pump with Change Data Capture:

- Restrictions on Using Oracle Data Pump with Change Data Capture
- Examples of Oracle Data Pump Export and Import Commands
- Publisher Considerations for Exporting and Importing Change Tables
- Re-Creating AutoLog Change Data Capture Objects After an Import Operation

Restrictions on Using Oracle Data Pump with Change Data Capture

Change Data Capture change sources, change sets, change tables, and subscriptions are exported and imported by the Oracle Data Pump expdp and impdp commands with the following restrictions.

- Change Data Capture objects are exported and imported only as part of full database export and import operations (those in which the expdp and impdb commands specify the FULL=y parameter). Schema-level import and export operations include some underlying objects (for example, the table underlying a change table), but not the Change Data Capture metadata needed for change data capture to occur.
- A user doing an import of Change Data Capture objects needs the following privileges:
  - either CREATE SEQUENCE or CREATE ANY SEQUENCE

    If the user doing the import is the original publisher, then CREATE SEQUENCE will suffice.
  - ALTER DATABASE
  - either EXP_FULL_DATABASE or IMP_FULL_DATABASE

    Access to the import package by PUBLIC has been removed and access is only to those that have either EXP_FULL_DATABASE or IMP_FULL_DATABASE.
- AutoLog change sources, change sets, and change tables are not supported. See "Re-Creating AutoLog Change Data Capture Objects After an Import Operation" on page 17-65.
- You should export asynchronous change sets and change tables at a time when users are not making DDL and DML changes to the database being exported.
- When importing asynchronous change sets and change tables, you must also import the underlying Oracle Streams configuration; set the Oracle Data Pump import parameter STREAMS_CONFIGURATION to y explicitly (or implicitly by

accepting the default), so that the necessary Streams objects are imported. If you perform an import operation and specify STREAMS_CONFIGURATION=n, then imported asynchronous change sets and change tables will not be able to continue capturing change data.

- Change Data Capture objects never overwrite existing objects when they are imported (similar to the effect of the import command TABLE_EXISTS_ACTION = skip parameter for tables). Change Data Capture generates warnings in the import log for these cases.
- Change Data Capture objects are validated at the end of an import operation to determine if all expected underlying objects are present in the correct form. Change Data Capture generates validation warnings in the import log if it detects validation problems. Imported Change Data Capture objects with validation warnings usually cannot continue capturing change data.

Examples of Oracle Data Pump Export and Import Commands

The following are examples of Data Pump export and import commands that support Change Data Capture objects:

```
> expdp DIRECTORY=dpump_dir FULL=y
> impdp DIRECTORY=dpump_dir FULL=y STREAMS_CONFIGURATION=y
```

See *Oracle Database Utilities* for information on Oracle Data Pump.

Publisher Considerations for Exporting and Importing Change Tables

The following are publisher considerations for exporting and importing change tables:

- When change tables are imported, the job queue is checked for a Change Data Capture purge job. If no purge job is found, then one is submitted automatically (using the DBMS_CDC_PUBLISH.PURGE procedure). If a change table is imported, but no subscriptions are taken out before the purge job runs (24 hours later, by default), then all rows in the table will be purged.

  The publisher can use one of the following methods to prevent the purging of data from a change table:

  – Suspend the purge job using the DBMS_SCHEDULER package to either disable the job (using the STOP_JOB procedure) or execute the job sometime in the future when there are subscriptions (using the SET_ATTRIBUTE procedure).

    Note: If you disable the purge job, you must remember to reset it once subscriptions have been activated. This prevents the change table from growing indefinitely.

  – Create a temporary subscription to preserve the change table data until real subscriptions appear. Then, drop the temporary subscription.

- When importing data into a source table for which a change table already exists, the imported data is also recorded in any associated change tables.

  Assume that the publisher has a source table SALES that has an associated change table ct_sales. When the publisher imports data into SALES, that data is also recorded in ct_sales.

- When importing a change table having the optional control ROW_ID column, the ROW_ID columns stored in the change table have meaning only if the associated source table has not been imported. If a source table is re-created or imported,

each row will have a new ROW_ID that is unrelated to the ROW_ID that was previously recorded in a change table.

The original level of export and import support available in Oracle9*i* Database is retained for backward compatibility. Synchronous change tables that reside in the SYNC_SET change set can be exported as part of a full database, schema, or individual table export operation and can be imported as needed. The following Change Data Capture objects are not included in the original export and import support: change sources, change sets, change tables that do not reside in the SYNC_SET change set, and subscriptions.

Re-Creating AutoLog Change Data Capture Objects After an Import Operation

After a Data Pump full database import operation completes for a database containing AutoLog Change Data Capture objects, the following steps must be performed to restore these objects:

1. The publisher must manually drop the database objects underlying AutoLog Change Data Capture objects.
2. The publisher must re-create the AutoLog change sources, change sets, and change tables using the appropriate DBMS_CDC_PUBLISH procedures.
3. Subscribers must re-create their subscriptions to the AutoLog change sets.

Change data may be lost in the interval between a Data Pump full database export operation involving AutoLog Change Data Capture objects and their re-creation after a Data Pump full database import operation in the preceding step. This can be minimized by preventing changes to the source tables during this interval, if possible.

Before re-creating an AutoLog Change Data Capture configuration after a Data Pump import operation, you must first drop the underlying objects: the table underlying a change table, subscriber views, a sequence used by the change set, and a Streams apply process, queue and queue table. Table 17–12 presents each underlying object and the method you use to drop it up after a Data Pump import operation.

Table 17–12 Methods to Drop Objects After a Data Pump Import Operation

Object	SQL Statement or PL/SQL Package to Drop Object
Table	SQL DROP TABLE statement
Subscriber View	SQL DROP VIEW statement
Sequence Used by Change Set	SQL DROP SEQUENCE statement
Streams Apply Process	PL/SQL DBMS_APPLY_ADM.DROP_APPLY() package
Streams Queue	PL/SQL DBMS_AQADM.DROP_QUEUE() package
Streams Queue Table	PL/SQL DBMS_AQADM.DROP_QUEUE_TABLE() package

You can obtain the name of the sequence used by a change set by querying the ALL_SEQUENCES view for a sequence name that begins with CDC$ and contains at least the initial characters of the change set name.

You can obtain the names of the Streams objects by querying the DBA_APPLY, DBA_QUEUES, and DBA_QUEUE_TABLES views for names that begin with CDC$ and contain at least the initial characters of the change set name.

Impact on Subscriptions When the Publisher Makes Changes

The Change Data Capture environment is dynamic. The publisher can add and drop change tables at any time. The publisher can also add columns to and drop columns from existing change tables at any time. The following list describes how changes to the Change Data Capture environment affect subscriptions:

- Subscribers do not get explicit notification if the publisher adds a new change table or adds columns to an existing change table. A subscriber can check the `ALL_PUBLISHED_COLUMNS` view to see if new columns have been added, and whether or not the subscriber has access to them.
- Table 17–13 describes what happens when the publisher adds a column to a change table.

Table 17–13 Effects of Publisher Adding a Column to a Change Table

If the publisher adds	And . . .	Then . . .
A user column	A new subscription includes this column	The subscription window for this subscription starts at the point the column was added.
A user column	A new subscription does not include this newly added column	The subscription window for this subscription starts at the earliest available change data. The new column will not be seen.
A user column	A subscription exists	The subscription window for this subscription remains unchanged.
A control column	A new subscription is created	The subscription window for this subscription starts at the earliest available change data. The subscription can see the control column immediately. All change table rows that existed prior to adding the control column will have the null value for the newly added control column.
A control column	A subscription exists	This subscription can see the new control columns after the subscription window is purged (`DBMS_CDC_PUBLISH.PURGE_WINDOW` procedure) such that the low boundary for the window crosses over the point when the control column was added.

Considerations for Synchronous Change Data Capture

The following sections provide information that the publisher should be aware of when using the synchronous mode of Change Data Capture:

- Restriction on Direct-Path INSERT
- Datatypes and Table Structures Supported for Synchronous Change Data Capture
- Limitation on Restoring Source Tables from the Recycle Bin

Restriction on Direct-Path INSERT

Synchronous Change Data Capture does not support the direct-path `INSERT` statement (and, by association, the `MERGE` statement and the `multi_table_insert` clause of the `INSERT` statement).

When the publisher creates a change table in synchronous mode, Change Data Capture creates triggers on the source table. Because a direct-path `INSERT` statement

disables all database triggers, any rows inserted into the source table using the SQL statement for direct-path INSERT in parallel DML mode will not be captured in the change table.

Similarly, Change Data Capture cannot capture the inserted rows from multitable insert and merge operations because these statements use a direct-path INSERT statement. The direct-path INSERT statement does not return an error message to indicate that the triggers used by Change Data Capture did not fire.

See *Oracle Database SQL Language Reference* for more information regarding the direct-path INSERT statement and triggers.

Datatypes and Table Structures Supported for Synchronous Change Data Capture

Synchronous Change Data Capture supports columns of all built-in Oracle datatypes except the following:

- BFILE
- BLOB
- CLOB
- LONG
- NCLOB
- ROWID
- UROWID
- object types (for example, XMLType)

In addition, virtual columns are not supported. Furthermore, Change Data Capture does not support the usage of quoted identifiers; as a consequence, the usage of reserved keywords with Change Data Capture is not supported. See *Oracle Database SQL Language Reference* for a list of reserved keywords.

Synchronous Change Data Capture does not support the following table structures:

- Source tables that are temporary tables
- Source tables that are object tables
- Index-organized tables with columns of unsupported datatypes (including LOB columns) or with overflow segments

Limitation on Restoring Source Tables from the Recycle Bin

If the source table for a synchronous change table is dropped and then restored from the recycle bin, changes are no longer captured in that change table. The publisher must create a new synchronous change table to capture future changes to the restored source table.

Considerations for Asynchronous Change Data Capture

The following sections provide information that the publisher and the source and staging database DBAs should be aware of when using the asynchronous mode of Change Data Capture:

- Asynchronous Change Data Capture and Redo Log Files
- Asynchronous Change Data Capture and Supplemental Logging

- Asynchronous Change Data Capture and Oracle Streams Components
- Datatypes and Table Structures Supported for Asynchronous Change Data Capture
- Restrictions for NOLOGGING and UNRECOVERABLE Operations

Asynchronous Change Data Capture and Redo Log Files

The asynchronous mode of Change Data Capture uses redo log files, as follows:

- HotLog

  Asynchronous HotLog and Distributed HotLog read the source database online redo log files whenever possible and the archived redo log files otherwise.

- AutoLog

  Asynchronous AutoLog Change Data Capture reads redo log files that have been copied from the source database to the staging database by redo transport services.

  When using the AutoLog online option, the destination attribute must be `LGWR ASYNC`. Redo transport services copies redo data from the online redo log at the source database to the standby redo log at the staging database. Change Data Capture obtains the change data after the source database transaction commits.

  When using the AutoLog archive option, the destination attribute can be `ARCH` or `LGWR ASYNC`. In `ARCH` mode, redo transport services copies archived redo log files to the staging database after a log switch occurs on the source database. In `LGWR` mode, redo transport services copies redo data to the staging database while it is being written to the online redo log file on the source database.

For log files to be archived, the source databases for asynchronous Change Data Capture must run in `ARCHIVELOG` mode, as specified with the following SQL statement:

```
ALTER DATABASE ARCHIVELOG;
```

See *Oracle Database Administrator's Guide* for information about running a database in `ARCHIVELOG` mode.

A redo log file used by Change Data Capture must remain available on the staging database until Change Data Capture has captured it. However, it is not necessary that the redo log file remain available until the Change Data Capture subscriber is done with the change data.

To determine which redo log files are no longer needed by Change Data Capture for a given change set, the publisher alters the change set's Streams capture process, which causes Streams to perform some internal cleanup and populates the `DBA_LOGMNR_PURGED_LOG` view. The publisher follows these steps:

1. Uses the following query on the staging database to get the three SCN values needed to determine an appropriate new `first_scn` value for the change set, `CHICAGO_DAILY`:

```
SELECT cap.CAPTURE_NAME, cap.FIRST_SCN, cap.APPLIED_SCN,
       cap.REQUIRED_CHECKPOINT_SCN
FROM DBA_CAPTURE cap, ALL_CHANGE_SETS cset
WHERE cset.SET_NAME = 'CHICAGO_DAILY' AND
      cap.CAPTURE_NAME = cset.CAPTURE_NAME;

CAPTURE_NAME                   FIRST_SCN APPLIED_SCN REQUIRED_CHECKPOINT_SCN
```

```
------------------------------ ---------- ----------- ----------------------
CDC$C_CHICAGO_DAILY                778059      778293                 778293
```

2. Determines a new `first_scn` value that is greater than or equal to the original `first_scn` value and less than or equal to the `applied_scn` and `required_checkpoint_scn` values returned by the query in step 1. In this example, this value is 778293, and the capture process name is `CDC$C_CHICAGO_DAILY`, therefore the publisher can alter the `first_scn` value for the capture process as follows:

```
BEGIN
DBMS_CAPTURE_ADM.ALTER_CAPTURE(
  capture_name => 'CDC$C_CHICAGO_DAILY',
  first_scn    => 778293);
END;
/
```

   If there is not an SCN value that meets these criteria, then the change set needs all of its redo log files.

3. Queries the `DBA_LOGMNR_PURGED_LOG` view to see any log files that are no longer needed by Change Data Capture:

```
SELECT FILE_NAME
FROM DBA_LOGMNR_PURGED_LOG;
```

> **Note:** Redo log files may be required on the staging database for purposes other than Change Data Capture. Before deleting a redo log file, the publisher should be sure that no other users need it.

See the information on setting the first SCN for an existing capture process and on capture process checkpoints in *Oracle Streams Concepts and Administration* for more information.

The `first_scn` value can be updated for all change sets in an AutoLog change source by using the `DBMS_CDC_PUBLISH.ALTER_AUTOLOG_CHANGE_SOURCE` `first_scn` parameter. Note that the new `first_scn` value must meet the criteria stated in Step 2 of the preceding list for all change sets in the AutoLog change source.

Both the size of the redo log files and the frequency with which a log switch occurs can affect the generation of the archived log files at the source database. For Change Data Capture, the most important factor in deciding what size to make a redo log file is the tolerance for latency between when a change is made and when that change data is available to subscribers. However, because the Oracle Database software attempts a check point at each log switch, if the redo log file is too small, frequent log switches will lead to frequent checkpointing and negatively impact the performance of the source database.

See *Oracle Data Guard Concepts and Administration* for step-by-step instructions on monitoring log file archival information. Substitute the terms source and staging database for the Oracle Data Guard terms primary database and archiving destinations, respectively.

When using redo transport services to supply redo log files to an AutoLog change source, gaps in the sequence of redo log files are automatically detected and resolved. If a situation arises where it is necessary to manually add a log file to an AutoLog change set, the publisher can use instructions on explicitly assigning log files to a downstream capture process described in *Oracle Streams Concepts and Administration*.

These instructions require the name of the capture process for the AutoLog change set. The publisher can obtain the name of the capture process for an AutoLog change set from the ALL_CHANGE_SETS data dictionary view.

Asynchronous Change Data Capture and Supplemental Logging

The asynchronous modes of Change Data Capture work best with appropriate supplemental logging on the source database. (Supplemental logging is not used by synchronous Change Data Capture.)

The source database DBA must enable some form of database-level supplemental logging. The following example enables minimal database-level supplemental logging:

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

In addition, Oracle recommends that the source database DBA:

- Supplementally log all source table columns that are part of a primary key or function to uniquely identify a row. This can be done using database-level or table-level identification key logging, or through a table-level unconditional log group.
- Create an unconditional log group for all source table columns that are captured by any asynchronous change table. This should be done before any change tables are created on a source table.

  ```
  ALTER TABLE sh.promotions
  ADD SUPPLEMENTAL LOG GROUP log_group_cust
  (PROMO_NAME, PROMO_SUBCATEGORY, PROMO_CATEGORY) ALWAYS;
  ```

  If an unconditional log group is not created for all source table columns to be captured, then when an update DML operation occurs, some unchanged user column values in change tables will be null instead of reflecting the actual source table value.

  For example, suppose a source table contains two columns, X and Y, and that the source database DBA has defined an unconditional log group for that table that includes only column Y. Furthermore, assume that a user updates only column Y in that table row. When the subscriber views the change data for that row, the value of the unchanged column X will be null. However, because the actual column value for X is excluded from the redo log file and therefore cannot be included in the change table, the subscriber cannot assume that the actual source table value for column X is null. The subscriber must rely on the contents of the TARGET_COLMAP$ control column to determine whether the actual source table value for column X is null or it is unchanged.

See *Oracle Database Utilities* for more information on the various types of supplemental logging.

Asynchronous Change Data Capture and Oracle Streams Components

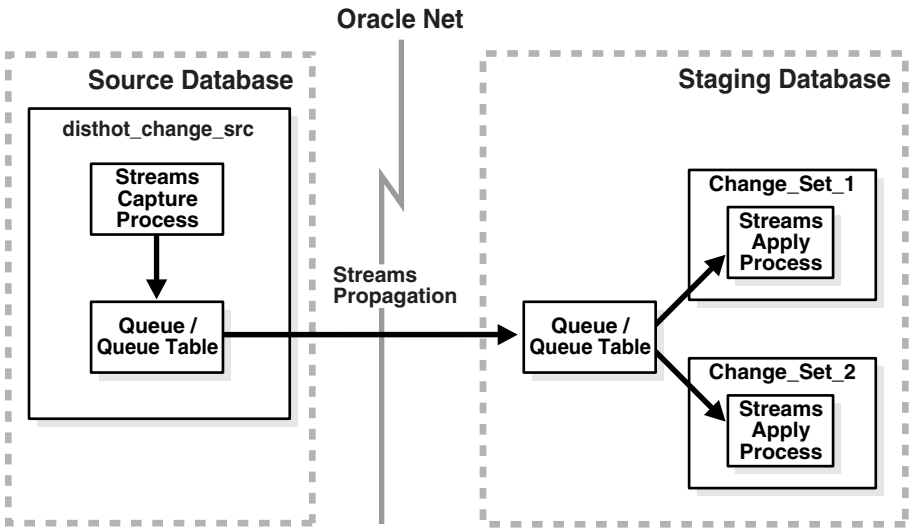
Asynchronous Change Data Capture generates components of Oracle Streams to capture change data and to populate change sets. These components must not be reused or shared for any other purposes. For example, the capture queue that Change Data Capture generates for a Distributed HotLog change source should not be used as the source queue for new user-created Streams propagations.

For HotLog and AutoLog modes, each change set contains a Streams capture process, queue, queue table and apply process. The staging database publisher owns all of these objects.

For Distributed HotLog mode, each change source resides on the source database and contains a Streams capture process, queue and queue table. Each change set resides on the staging database and contains a Streams apply process. When the first change set in a change source is created, Change Data Capture generates a queue and queue table on the staging database to be shared among all change sets in the change source. Change Data Capture also generates a Streams Propagation on the source database from the change source queue to the change set queue on the staging database. The source database publisher owns the source database objects and the staging database publisher owns the staging database objects.

Figure 17–8 illustrates an example of how Streams components are used in an asynchronous Distributed HotLog Change Data Capture environment. In this example, there is a Distributed HotLog change source `disthot_change_src` on the source database and two change sets, `Change_Set_1` and `Change_Set_2`, on the staging database.

Figure 17–8 Streams Components in an Asynchronous Distributed HotLog Change Data Capture System



Datatypes and Table Structures Supported for Asynchronous Change Data Capture

Asynchronous Change Data Capture supports columns of all built-in Oracle datatypes except the following:

- `BFILE`
- `LONG`
- `ROWID`
- `UROWID`
- object types (for example, XMLType)

In addition, virtual columns are not supported. Furthermore, Change Data Capture does not support the usage of quoted identifiers; as a consequence, the usage of reserved keywords with Change Data Capture is not supported. See *Oracle Database SQL Language Reference* for a list of reserved keywords.

Asynchronous Change Data Capture does not support the following table structures:

- Source tables that are temporary tables
- Source tables that are object tables
- Index-organized tables with columns of unsupported datatypes (including LOB columns) or with overflow segments

Restrictions for NOLOGGING and UNRECOVERABLE Operations

If you use the NOLOGGING or UNRECOVERABLE keyword for a SQL operation, asynchronous Change Data Capture cannot capture the changes from that operation. Similarly, if you use the UNRECOVERABLE clause in the control file for a SQL*Loader direct path load, then the changes from that direct load cannot be captured by asynchronous Change Data Capture.

See *Oracle Database SQL Language Reference* for information about the NOLOGGING and UNRECOVERABLE keywords and *Oracle Database Utilities* for information about direct path loads and SQL*Loader.

Implementation and System Configuration

Change Data Capture comes packaged with the appropriate Oracle drivers already installed with which you can implement either asynchronous or synchronous data capture. The synchronous mode of Change Data Capture is included with the Standard Edition, but the asynchronous mode requires you have the Enterprise Edition.

In addition, note that Change Data Capture uses Java. Therefore, when you install Oracle Database, ensure that Java is enabled.

Change Data Capture places system triggers on the SQL CREATE TABLE, ALTER TABLE, and DROP TABLE statements. If system triggers are disabled on the source database, Change Data Capture will not function correctly. Therefore, you should never disable system triggers.

To remove Change Data Capture from the database, the SQL script rmcdc.sql is provided in the admin directory. This will remove the system triggers that Change Data Capture places on the SQL CREATE TABLE, ALTER TABLE, and DROP TABLE statements. In addition, rmcdc.sql removes all Java classes used by Change Data Capture. Note that after rmcdc.sql is called, Change Data Capture will no longer operate on the system. If the system administrator decides to remove the Java Virtual Machine from a database, rmcdc.sql must be called before rmjvm is called.

To reinstall Change Data Capture, the SQL script initcdc.sql is provided in the admin directory. It creates the Change Data Capture system triggers and Java classes that are required by Change Data Capture. Note that the Java Virtual Machine must be available to reinstall Change Data Capture.

Database Configuration Assistant Considerations

All of the predefined templates provided with the Database Configuration Assistant support the Oracle Change Data Capture feature. The predefined templates are:

- General Purpose
- Data Warehouse
- Transaction Processing
- New Database

If you choose the New Database option to build a custom database, note that Oracle JVM is selected by default in the Additional database configurations dialog box. Do not change this setting; Oracle Change Data Capture requires the Oracle JVM feature.

Summary of Supported Distributed HotLog Configurations and Restrictions

The following sections describe the supported configurations for the Distributed HotLog mode of Change Data Capture as well as the restrictions.

Oracle Database Releases for Source and Staging Databases

The Distributed HotLog mode of Change Data Capture allows the following combinations of Oracle Database releases for the source and staging databases:

- The source database can be Oracle Database release 9.2 (9.2.0.6 or higher patch set), 10.1, 10.2, 11.1, or 11.2.
- The staging database can be Oracle Database release 11.1 or 11.2.

Note: When a release 9.2 or 10.1 Oracle Database serves as the source database for the Distributed HotLog mode of Change Data Capture, metadata for the source database is stored on the staging database, whereas metadata for an Oracle Database release 11.1 or 11.2 source database is stored on the source database. Therefore, to view the metadata for an Oracle Database release 9.2 or 10.1 source database, the publisher must query the `CHANGE_SOURCES` data dictionary view on the staging database, and to view the metadata for an Oracle Database release 11.1 or 11.2 source database, the publisher must query the `CHANGE_SOURCES` data dictionary view on the source database.

Upgrading a Distributed HotLog Change Source to Oracle Release 11 (11.1 or 11.2)

As mentioned in the previous topic, the metadata for an Oracle Database release 11 Distributed HotLog change source is stored on the source database, but the metadata for a release 9.2 or 10.1 change source is stored on the staging database.

When you upgrade a release 10.2 Oracle Database to release 11, then there is additional metadata movement required.

When you upgrade a release 9.2 or 10.1 Oracle Database to release 11, Change Data Capture does not move the source database metadata from the staging database to the source database as part of the upgrade operation. However, the first time the change source is enabled after the upgrade (using the `DBMS_CDC_PUBLISH.ALTER_HOTLOG_CHANGE_SOURCE` procedure), Change Data Capture detects that the metadata needs to be upgraded and moved to the source database and does so automatically.

If the publisher prefers to not alter the change source immediately after an upgrade, the change source metadata can remain on the staging database until a more appropriate time. Oracle recommends that the source database metadata not be left on the staging database indefinitely after an upgrade to Oracle Database release 11 so that information about the Distributed HotLog change source becomes available in the data dictionary views on the source database.

Hardware Platforms and Operating Systems

The Distributed HotLog mode of Change Data Capture supports the use of different hardware platforms or operating systems (or both) for the source and staging databases.

Requirements for Multiple Publishers on the Staging Database

If there are multiple publishers on the staging database for the Distributed HotLog mode of Change Data capture, and one publisher defines a change table in another publisher's Distributed HotLog change set, then Change Data Capture uses the database link established by publisher who created the change set to access the source database. Therefore, the database link to the source database established by the publisher who created the change set must be intact for the change table to be successfully created. If the change set publisher's database link is not present when creating a change table, an error is returned indicating that the connection description for the remote database was not found.

Requirements for Database Links

The database link from the source database to the staging database must exist for the life of a Distributed HotLog change source.

The database link from the staging database to the source database must exist when creating, altering or dropping Distributed HotLog change sources, change sets and change tables. However, this database link is not required for change capture to occur. Once the required Distributed HotLog change sources, change sets and change tables are in place and enabled, this database link can be dropped without interrupting change capture. This database link would need to be re-created to create, alter or drop Distributed HotLog change sources, change sets and change tables.

Part V

Data Warehouse Performance

This section deals with ways to improve your data warehouse's performance, and contains the following chapters:

18

Basic Query Rewrite

This chapter discusses query rewrite in Oracle, and contains:

Overview of Query Rewrite

When base tables contain large amount of data, it is expensive and time-consuming to compute the required aggregates or to compute joins between these tables. In such cases, queries can take minutes or even hours. Because materialized views contain already precomputed aggregates and joins, Oracle Database employs an extremely powerful process called query rewrite to quickly answer the query using materialized views.

One of the major benefits of creating and maintaining materialized views is the ability to take advantage of query rewrite, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped just like indexes without invalidating the SQL in the application code.

A query undergoes several checks to determine whether it is a candidate for query rewrite. If the query fails any of the checks, then the query is applied to the detail tables rather than the materialized view. This can be costly in terms of response time and processing power.

The optimizer uses two different methods to recognize when to rewrite a query in terms of a materialized view. The first method is based on matching the SQL text of the query with the SQL text of the materialized view definition. If the first method fails, the optimizer uses the more general method in which it compares joins, selections, data columns, grouping columns, and aggregate functions between the query and materialized views.

Query rewrite operates on queries and subqueries in the following types of SQL statements:

- `SELECT`
- `CREATE TABLE ... AS SELECT`
- `INSERT INTO ... SELECT`

It also operates on subqueries in the set operators UNION, UNION ALL, INTERSECT, and MINUS, and subqueries in DML statements such as INSERT, DELETE, and UPDATE.

Dimensions, constraints, and rewrite integrity levels affect whether or not a given query is rewritten to use one or more materialized views. Additionally, query rewrite can be enabled or disabled by REWRITE and NOREWRITE hints and the QUERY_REWRITE_ENABLED session parameter.

The DBMS_MVIEW.EXPLAIN_REWRITE procedure advises whether query rewrite is possible on a query and, if so, which materialized views are used. It also explains why a query cannot be rewritten.

When Does Oracle Rewrite a Query?

A query is rewritten only when a certain number of conditions are met:

- Query rewrite must be enabled for the session.
- A materialized view must be enabled for query rewrite.
- The rewrite integrity level should allow the use of the materialized view. For example, if a materialized view is not fresh and query rewrite integrity is set to ENFORCED, then the materialized view is not used.
- Either all or part of the results requested by the query must be obtainable from the precomputed result stored in the materialized view or views.

To test these conditions, the optimizer may depend on some of the data relationships declared by the user using constraints and dimensions, among others, hierarchies, referential integrity, and uniqueness of key data, and so on.

Ensuring that Query Rewrite Takes Effect

You must follow several conditions to enable query rewrite:

1. Individual materialized views must have the ENABLE QUERY REWRITE clause.
2. The session parameter QUERY_REWRITE_ENABLED must be set to TRUE (the default) or FORCE.
3. Cost-based optimization must be used by setting the initialization parameter OPTIMIZER_MODE to ALL_ROWS, FIRST_ROWS, or FIRST_ROWS_*n*.

If step 1 has not been completed, a materialized view is never eligible for query rewrite. You can specify ENABLE QUERY REWRITE either with the ALTER MATERIALIZED VIEW statement or when the materialized view is created, as illustrated in the following:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id = p.prod_id;
```

The NOREWRITE hint disables query rewrite in a SQL statement, overriding the QUERY_REWRITE_ENABLED parameter, and the REWRITE hint (when used with mv_name) restricts the eligible materialized views to those named in the hint.

You can use the DBMS_ADVISOR.TUNE_MVIEW procedure to optimize a CREATE MATERIALIZED VIEW statement to enable general QUERY REWRITE.

Initialization Parameters for Query Rewrite

The following three initialization parameter settings control query rewrite behavior:

- OPTIMIZER_MODE = ALL_ROWS (default), FIRST_ROWS, or FIRST_ROWS_*n*

  With OPTIMIZER_MODE set to FIRST_ROWS, the optimizer uses a mix of costs and heuristics to find a best plan for fast delivery of the first few rows. When set to FIRST_ROWS_*n*, the optimizer uses a cost-based approach and optimizes with a goal of best response time to return the first *n* rows (where n = 1, 10, 100, 1000).

- QUERY_REWRITE_ENABLED = TRUE (default), FALSE, or FORCE

  This option enables the query rewrite feature of the optimizer, enabling the optimizer to utilize materialized views to enhance performance. If set to FALSE, this option disables the query rewrite feature of the optimizer and directs the optimizer not to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower.

  If set to FORCE, this option enables the query rewrite feature of the optimizer and directs the optimizer to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower.

- QUERY_REWRITE_INTEGRITY

  This parameter is optional, but must be set to STALE_TOLERATED, TRUSTED, or ENFORCED (the default) if it is specified (see "Accuracy of Query Rewrite" on page 18-3).

  By default, the integrity level is set to ENFORCED. In this mode, all constraints must be validated. Therefore, if you use ENABLE NOVALIDATE RELY, certain types of query rewrite might not work. To enable query rewrite in this environment (where constraints have not been validated), you should set the integrity level to a lower level of granularity such as TRUSTED or STALE_TOLERATED.

Controlling Query Rewrite

A materialized view is only eligible for query rewrite if the ENABLE QUERY REWRITE clause has been specified, either initially when the materialized view was first created or subsequently with an ALTER MATERIALIZED VIEW statement.

You can set the session parameters described previously for all sessions using the ALTER SYSTEM SET statement or in the initialization file. For a given user's session, ALTER SESSION can be used to disable or enable query rewrite for that session only. An example is the following:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

You can set the level of query rewrite for a session, thus allowing different users to work at different integrity levels. The possible statements are:

```
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = STALE_TOLERATED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = TRUSTED;
ALTER SESSION SET QUERY_REWRITE_INTEGRITY = ENFORCED;
```

Accuracy of Query Rewrite

Query rewrite offers three levels of rewrite integrity that are controlled by the session parameter QUERY_REWRITE_INTEGRITY, which can either be set in your parameter file or controlled using an ALTER SYSTEM or ALTER SESSION statement. The three values are as follows:

- ENFORCED

  This is the default mode. The optimizer only uses fresh data from the materialized views and only use those relationships that are based on ENABLED VALIDATED primary, unique, or foreign key constraints.

- TRUSTED

  In TRUSTED mode, the optimizer trusts that the relationships declared in dimensions and RELY constraints are correct. In this mode, the optimizer also uses prebuilt materialized views or materialized views based on views, and it uses relationships that are not enforced as well as those that are enforced. It also trusts declared but not ENABLED VALIDATED primary or unique key constraints and data relationships specified using dimensions. This mode offers greater query rewrite capabilities but also creates the risk of incorrect results if any of the trusted relationships you have declared are incorrect.

- STALE_TOLERATED

  In STALE_TOLERATED mode, the optimizer uses materialized views that are valid but contain stale data as well as those that contain fresh data. This mode offers the maximum rewrite capability but creates the risk of generating inaccurate results.

If rewrite integrity is set to the safest level, ENFORCED, the optimizer uses only enforced primary key constraints and referential integrity constraints to ensure that the results of the query are the same as the results when accessing the detail tables directly. If the rewrite integrity is set to levels other than ENFORCED, there are several situations where the output with rewrite can be different from that without it:

- A materialized view can be out of synchronization with the master copy of the data. This generally happens because the materialized view refresh procedure is pending following bulk load or DML operations to one or more detail tables of a materialized view. At some data warehouse sites, this situation is desirable because it is not uncommon for some materialized views to be refreshed at certain time intervals.
- The relationships implied by the dimension objects are invalid. For example, values at a certain level in a hierarchy do not roll up to exactly one parent value.
- The values stored in a prebuilt materialized view table might be incorrect.
- A wrong answer can occur because of bad data relationships defined by unenforced table or view constraints.

Privileges for Enabling Query Rewrite

Use of a materialized view is based not on privileges the user has on that materialized view, but on the privileges the user has on detail tables or views in the query.

The system privilege GRANT QUERY REWRITE lets you enable materialized views in your own schema for query rewrite only if all tables directly referenced by the materialized view are in that schema. The GRANT GLOBAL QUERY REWRITE privilege enables you to enable materialized views for query rewrite even if the materialized view references objects in other schemas. Alternatively, you can use the QUERY REWRITE object privilege on tables and views outside your schema.

The privileges for using materialized views for query rewrite are similar to those for definer's rights procedures.

Sample Schema and Materialized Views

The following sections use the sh sample schema and a few materialized views to illustrate how the optimizer uses data relationships to rewrite queries.

The query rewrite examples in this chapter mainly refer to the following materialized views. These materialized views do not necessarily represent the most efficient implementation for the sh schema. Instead, they are a base for demonstrating rewrite capabilities. Further examples demonstrating specific functionality can be found throughout this chapter.

The following materialized views contain joins and aggregates:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;

CREATE MATERIALIZED VIEW sum_sales_prod_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;

CREATE MATERIALIZED VIEW sum_sales_pscat_month_city_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
       COUNT(s.amount_sold) AS count_amount_sold
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id AND s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

The following materialized views contain joins only:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW join_sales_time_product_oj_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id(+);
```

Although it is not a strict requirement, it is highly recommended that you collect statistics on the materialized views so that the optimizer can determine whether to rewrite the queries. You can do this either on a per-object base or for all newly created objects without statistics. The following is an example of a per-object base, shown for join_sales_time_product_mv:

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS ( -
   'SH','JOIN_SALES_TIME_PRODUCT_MV', estimate_percent   => 20, -
```

```
    block_sample   => TRUE, cascade   => TRUE);
```

The following illustrates a statistics collection for all newly created objects without statistics:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS ( 'SH', -
   options            => 'GATHER EMPTY', -
   estimate_percent   =>  20, block_sample  =>  TRUE, -
   cascade            =>  TRUE);
```

How to Verify Query Rewrite Occurred

Because query rewrite occurs transparently, special steps have to be taken to verify that a query has been rewritten. Of course, if the query runs faster, this should indicate that rewrite has occurred, but that is not proof. Therefore, to confirm that query rewrite does occur, use the `EXPLAIN PLAN` statement or the `DBMS_MVIEW.EXPLAIN_REWRITE` procedure. See "Verifying that Query Rewrite has Occurred" on page 19-60 for further information.

Example of Query Rewrite

Consider the following materialized view, `cal_month_sales_mv`, which provides an aggregation of the dollar amount sold in every month:

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Let us say that, in a typical month, the number of sales in the store is around one million. So this materialized aggregate view has the precomputed aggregates for the dollar amount sold for each month. Now consider the following query, which asks for the sum of the amount sold at the store for each calendar month:

```
SELECT t.calendar_month_desc, SUM(s.amount_sold)
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

In the absence of the previous materialized view and query rewrite feature, Oracle will have to access the `sales` table directly and compute the sum of the amount sold to return the results. This involves reading many million rows from the `sales` table which will invariably increase the query response time due to the disk access. The join in the query will also further slow down the query response as the join needs to be computed on many million rows. In the presence of the materialized view `cal_month_sales_mv`, query rewrite will transparently rewrite the previous query into the following query:

```
SELECT calendar_month, dollars
FROM cal_month_sales_mv;
```

Because there are only a few dozens rows in the materialized view `cal_month_sales_mv` and no joins, Oracle Database returns the results instantly. This simple example illustrates the power of query rewrite with materialized views!

19

Advanced Query Rewrite

This chapter discusses advanced query rewrite topics in Oracle, and contains:

- How Oracle Rewrites Queries
- Types of Query Rewrite
- Other Query Rewrite Considerations
- Advanced Query Rewrite Using Equivalences
- Creating Result Cache Materialized Views with Equivalences
- Verifying that Query Rewrite has Occurred
- Design Considerations for Improving Query Rewrite Capabilities

How Oracle Rewrites Queries

The optimizer uses a number of different methods to rewrite a query. The first step in determining whether query rewrite is possible is to see if the query satisfies the following prerequisites:

- Joins present in the materialized view are present in the SQL.
- There is sufficient data in the materialized view(s) to answer the query.

After that, it must determine how it will rewrite the query. The simplest case occurs when the result stored in a materialized view exactly matches what is requested by a query. The optimizer makes this type of determination by comparing the text of the query with the text of the materialized view definition. This text match method is most straightforward but the number of queries eligible for this type of query rewrite is minimal.

When the text comparison test fails, the optimizer performs a series of generalized checks based on the joins, selections, grouping, aggregates, and column data fetched. This is accomplished by individually comparing various clauses (`SELECT`, `FROM`, `WHERE`, `HAVING`, or `GROUP BY`) of a query with those of a materialized view.

This section discusses the optimizer in more detail, as well as the following types of query rewrite:

- Text Match Rewrite
- General Query Rewrite Methods

Cost-Based Optimization

When a query is rewritten, Oracle's cost-based optimizer compares the cost of the rewritten query and original query and chooses the cheaper execution plan.

Query rewrite is available with cost-based optimization. Oracle Database optimizes the input query with and without rewrite and selects the least costly alternative. The optimizer rewrites a query by rewriting one or more query blocks, one at a time.

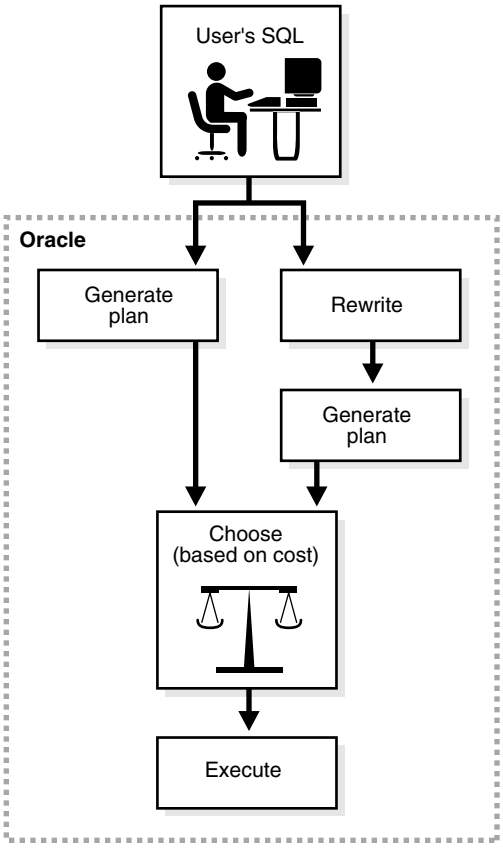
If query rewrite has a choice between several materialized views to rewrite a query block, it selects the ones which can result in reading in the least amount of data. After a materialized view has been selected for a rewrite, the optimizer then tests whether the rewritten query can be rewritten further with other materialized views. This process continues until no further rewrites are possible. Then the rewritten query is optimized and the original query is optimized. The optimizer compares these two optimizations and selects the least costly alternative.

Because optimization is based on cost, it is important to collect statistics both on tables involved in the query and on the tables representing materialized views. Statistics are fundamental measures, such as the number of rows in a table, that are used to calculate the cost of a rewritten query. They are created by using the `DBMS_STATS` package.

Queries that contain inline or named views are also candidates for query rewrite. When a query contains a named view, the view name is used to do the matching between a materialized view and the query. When a query contains an inline view, the inline view can be merged into the query before matching between a materialized view and the query occurs.

Figure 19–1 presents a graphical view of the cost-based approach used during the rewrite process.

Figure 19–1 The Query Rewrite Process



General Query Rewrite Methods

The optimizer has a number of different types of query rewrite methods that it can choose from to answer a query. When text match rewrite is not possible, this group of rewrite methods is known as general query rewrite. The advantage of using these more advanced techniques is that one or more materialized views can be used to answer a number of different queries and the query does not always have to match the materialized view exactly for query rewrite to occur.

When using general query rewrite methods, the optimizer uses data relationships on which it can depend, such as primary and foreign key constraints and dimension objects. For example, primary key and foreign key relationships tell the optimizer that each row in the foreign key table joins with at most one row in the primary key table. Furthermore, if there is a `NOT NULL` constraint on the foreign key, it indicates that each row in the foreign key table must join to exactly one row in the primary key table. A dimension object describes the relationship between, say, day, months, and year, which can be used to roll up data from the day to the month level.

Data relationships such as these are very important for query rewrite because they tell what type of result is produced by joins, grouping, or aggregation of data. Therefore, to maximize the rewritability of a large set of queries when such data relationships exist in a database, you should declare constraints and dimensions.

When are Constraints and Dimensions Needed?

Table 19–1 illustrates when dimensions and constraints are required for different types of query rewrite. These types of query rewrite are described throughout this chapter.

Table 19–1 Dimension and Constraint Requirements for Query Rewrite

Query Rewrite Types	Dimensions		Primary Key/Foreign Key/Not Null Constraints
Matching SQL Text	Not Required		Not Required
Join Back	Required	OR	Required
Aggregate Computability	Not Required		Not Required
Aggregate Rollup	Not Required		Not Required
Rollup Using a Dimension	Required		Not Required
Filtering the Data	Not Required		Not Required
PCT Rewrite	Not Required		Not Required
Multiple Materialized Views	Not Required		Not Required

Checks Made by Query Rewrite

For query rewrite to occur, there are a number of checks that the data must pass. These checks are:

- Join Compatibility Check
- Data Sufficiency Check
- Grouping Compatibility Check
- Aggregate Computability Check

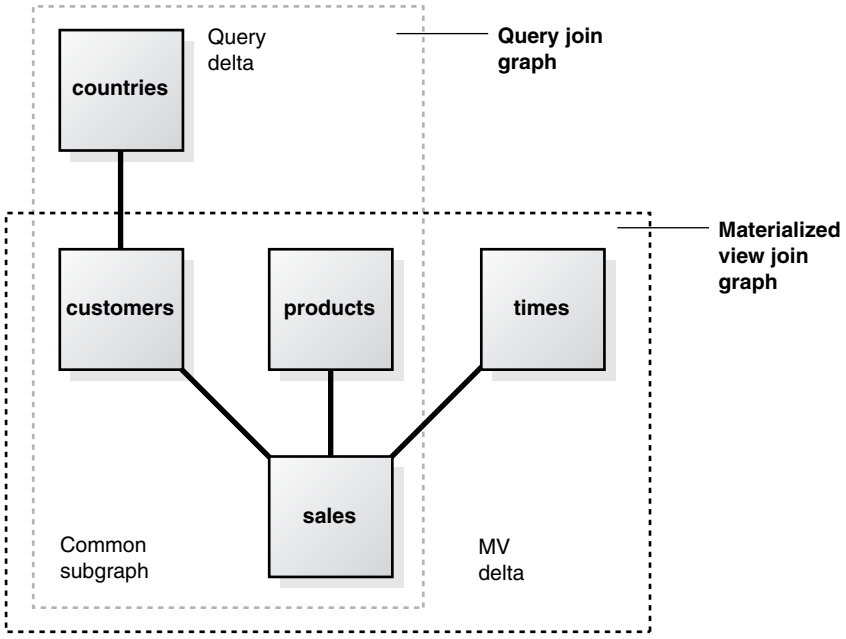
Join Compatibility Check

In this check, the joins in a query are compared against the joins in a materialized view. In general, this comparison results in the classification of joins into three categories:

- Common joins that occur in both the query and the materialized view. These joins form the common subgraph.
- Delta joins that occur in the query but not in the materialized view. These joins form the query delta subgraph.
- Delta joins that occur in the materialized view but not in the query. These joins form the materialized view delta subgraph.

These can be visualized as shown in Figure 19–2.

Figure 19–2 Query Rewrite Subgraphs



Common Joins The common join pairs between the two must be of the same type, or the join in the query must be derivable from the join in the materialized view. For example, if a materialized view contains an outer join of table `A` with table `B`, and a query contains an inner join of table `A` with table `B`, the result of the inner join can be derived by filtering the antijoin rows from the result of the outer join. For example, consider the following query:

```
SELECT p.prod_name, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id
AND mv.week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                           AND TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY p.prod_name, mv.week_ending_day;
```

The common joins between this query and the materialized view `join_sales_time_product_mv` are:

```
s.time_id = t.time_id AND s.prod_id = p.prod_id
```

They match exactly and the query can be rewritten as follows:

```
SELECT p.prod_name, mv.week_ending_day, SUM(s.amount_sold)
FROM   join_sales_time_product_mv
WHERE  mv.week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
                          AND   TO_DATE('10-AUG-1999','DD-MON-YYYY')
GROUP BY mv.prod_name, mv.week_ending_day;
```

The query could also be answered using the `join_sales_time_product_oj_mv` materialized view where inner joins in the query can be derived from outer joins in the materialized view. The rewritten version (transparently to the user) filters out the antijoin rows. The rewritten query has the following structure:

```
SELECT mv.prod_name, mv.week_ending_day, SUM(mv.amount_sold)
FROM   join_sales_time_product_oj_mv mv
WHERE  mv.week_ending_day BETWEEN TO_DATE('01-AUG-1999','DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999','DD-MON-YYYY') AND mv.prod_id IS NOT NULL
```

```
GROUP BY mv.prod_name, mv.week_ending_day;
```

In general, if you use an outer join in a materialized view containing only joins, you should put in the materialized view either the primary key or the rowid on the right side of the outer join. For example, in the previous example, `join_sales_time_product_oj_mv`, there is a primary key on both `sales` and `products`.

Another example of when a materialized view containing only joins is used is the case of a semijoin rewrites. That is, a query contains either an `EXISTS` or an `IN` subquery with a single table. Consider the following query, which reports the products that had sales greater than $1,000:

```
SELECT DISTINCT p.prod_name
FROM products p
WHERE EXISTS (SELECT p.prod_id, SUM(s.amount_sold) FROM sales s
              WHERE p.prod_id=s.prod_id HAVING SUM(s.amount_sold) > 1000)
              GROUP BY p.prod_id);
```

This query could also be represented as:

```
SELECT DISTINCT p.prod_name
FROM products p WHERE p.prod_id IN (SELECT s.prod_id FROM sales s
                   WHERE s.amount_sold > 1000);
```

This query contains a semijoin (`s.prod_id = p.prod_id`) between the `products` and the `sales` table.

This query can be rewritten to use either the `join_sales_time_product_mv` materialized view, if foreign key constraints are active or `join_sales_time_product_oj_mv` materialized view, if primary keys are active. Observe that both materialized views contain `s.prod_id=p.prod_id`, which can be used to derive the semijoin in the query. The query is rewritten with `join_sales_time_product_mv` as follows:

```
SELECT mv.prod_name
FROM (SELECT DISTINCT mv.prod_name FROM  join_sales_time_product_mv mv
      WHERE mv.amount_sold > 1000);
```

If the materialized view `join_sales_time_product_mv` is partitioned by `time_id`, then this query is likely to be more efficient than the original query because the original join between `sales` and `products` has been avoided. The query could be rewritten using `join_sales_time_product_oj_mv` as follows.

```
SELECT mv.prod_name
FROM (SELECT DISTINCT mv.prod_name FROM join_sales_time_product_oj_mv mv
      WHERE mv.amount_sold > 1000 AND mv.prod_id IS NOT NULL);
```

Rewrites with semi-joins are restricted to materialized views with joins only and are not possible for materialized views with joins and aggregates.

Query Delta Joins A **query delta join** is a join that appears in the query but not in the materialized view. Any number and type of delta joins in a query are allowed and they are simply retained when the query is rewritten with a materialized view. In order for the retained join to work, the materialized view must contain the joining key. Upon rewrite, the materialized view is joined to the appropriate tables in the query delta. For example, consider the following query:

```
SELECT p.prod_name, t.week_ending_day, c.cust_city, SUM(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id
AND    s.cust_id = c.cust_id
```

```
GROUP BY p.prod_name, t.week_ending_day, c.cust_city;
```

Using the materialized view join_sales_time_product_mv, common joins are: s.time_id=t.time_id and s.prod_id=p.prod_id. The delta join in the query is s.cust_id=c.cust_id. The rewritten form then joins the join_sales_time_product_mv materialized view with the customers table as follows:

```
SELECT mv.prod_name, mv.week_ending_day, c.cust_city, SUM(mv.amount_sold)
FROM   join_sales_time_product_mv mv, customers c
WHERE  mv.cust_id = c.cust_id
GROUP BY mv.prod_name, mv.week_ending_day, c.cust_city;
```

Materialized View Delta Joins A **materialized view delta join** is a join that appears in the materialized view but not the query. All delta joins in a materialized view are required to be lossless with respect to the result of common joins. A lossless join guarantees that the result of common joins is not restricted. A **lossless** join is one where, if two tables called A and B are joined together, rows in table A will always match with rows in table B and no data will be lost, hence the term lossless join. For example, every row with the foreign key matches a row with a primary key provided no nulls are allowed in the foreign key. Therefore, to guarantee a lossless join, it is necessary to have FOREIGN KEY, PRIMARY KEY, and NOT NULL constraints on appropriate join keys. Alternatively, if the join between tables A and B is an outer join (A being the outer table), it is lossless as it preserves all rows of table A.

All delta joins in a materialized view are required to be non-duplicating with respect to the result of common joins. A non-duplicating join guarantees that the result of common joins is not duplicated. For example, a non-duplicating join is one where, if table A and table B are joined together, rows in table A will match with at most one row in table B and no duplication occurs. To guarantee a non-duplicating join, the key in table B must be constrained to unique values by using a primary key or unique constraint.

Consider the following query that joins sales and times:

```
SELECT t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, times t
WHERE s.time_id = t.time_id AND t.week_ending_day BETWEEN TO_DATE
 ('01-AUG-1999', 'DD-MON-YYYY') AND TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The materialized view join_sales_time_product_mv has an additional join (s.prod_id=p.prod_id) between sales and products. This is the delta join in join_sales_time_product_mv. You can rewrite the query if this join is lossless and non-duplicating. This is the case if s.prod_id is a foreign key to p.prod_id and is not null. The query is therefore rewritten as:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
                       AND     TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The query can also be rewritten with the materialized view join_sales_time_product_mv_oj where foreign key constraints are not needed. This view contains an outer join (s.prod_id=p.prod_id(+)) between sales and products. This makes the join lossless. If p.prod_id is a primary key, then the non-duplicating condition is satisfied as well and optimizer rewrites the query as follows:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
```

```
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

The query can also be rewritten with the materialized view `join_sales_time_product_mv_oj` where foreign key constraints are not needed. This view contains an outer join (`s.prod_id=p.prod_id(+)`) between `sales` and `products`. This makes the join lossless. If `p.prod_id` is a primary key, then the non-duplicating condition is satisfied as well and optimizer rewrites the query as follows:

```
SELECT week_ending_day, SUM(amount_sold)
FROM   join_sales_time_product_oj_mv
WHERE  week_ending_day BETWEEN TO_DATE('01-AUG-1999', 'DD-MON-YYYY')
  AND  TO_DATE('10-AUG-1999', 'DD-MON-YYYY')
GROUP BY week_ending_day;
```

Note that the outer join in the definition of `join_sales_time_product_mv_oj` is not necessary because the primary key - foreign key relationship between `sales` and `products` in the `sh` schema is already lossless. It is used for demonstration purposes only, and would be necessary if `sales.prod_id` were nullable, thus violating the losslessness of the join condition `sales.prod_id = products.prod_id`.

Current limitations restrict most rewrites with outer joins to materialized views with joins only. There is limited support for rewrites with materialized aggregate views with outer joins, so those materialized views should rely on foreign key constraints to assure losslessness of materialized view delta joins.

Join Equivalence Recognition Query rewrite is able to make many transformations based upon the recognition of equivalent joins. Query rewrite recognizes the following construct as being equivalent to a join:

```
WHERE table1.column1 = F(args)   /* sub-expression A */
AND table2.column2 = F(args)     /* sub-expression B */
```

If `F(args)` is a PL/SQL function that is declared to be deterministic and the arguments to both invocations of `F` are the same, then the combination of subexpression `A` with subexpression `B` be can be recognized as a join between `table1.column1` and `table2.column2`. That is, the following expression is equivalent to the previous expression:

```
WHERE table1.column1 = F(args)          /* sub-expression A */
AND table2.column2 = F(args)            /* sub-expression B */
AND table1.column1 = table2.column2     /* join-expression J */
```

Because join-expression `J` can be inferred from sub-expression `A` and subexpression `B`, the inferred join can be used to match a corresponding join of `table1.column1 = table2.column2` in a materialized view.

Data Sufficiency Check

In this check, the optimizer determines if the necessary column data requested by a query can be obtained from a materialized view. For this, the equivalence of one column with another is used. For example, if an inner join between table `A` and table `B` is based on a join predicate `A.X = B.X`, then the data in column `A.X` equals the data in column `B.X` in the result of the join. This data property is used to match column `A.X` in a query with column `B.X` in a materialized view or vice versa. For example, consider the following query:

```
SELECT p.prod_name, s.time_id, t.week_ending_day, SUM(s.amount_sold)
FROM sales s, products p, times t
```

```
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id
GROUP BY p.prod_name, s.time_id, t.week_ending_day;
```

This query can be answered with `join_sales_time_product_mv` even though the materialized view does not have `s.time_id`. Instead, it has `t.time_id`, which, through a join condition `s.time_id=t.time_id`, is equivalent to `s.time_id`. Thus, the optimizer might select the following rewrite:

```
SELECT prod_name, time_id, week_ending_day, SUM(amount_sold)
FROM join_sales_time_product_mv
GROUP BY prod_name, time_id, week_ending_day;
```

Grouping Compatibility Check

This check is required only if both the materialized view and the query contain a `GROUP BY` clause. The optimizer first determines if the grouping of data requested by a query is exactly the same as the grouping of data stored in a materialized view. In other words, the level of grouping is the same in both the query and the materialized view. If the materialized views groups on all the columns and expressions in the query and also groups on additional columns or expressions, query rewrite can reaggregate the materialized view over the grouping columns and expressions of the query to derive the same result requested by the query.

Aggregate Computability Check

This check is required only if both the query and the materialized view contain aggregates. Here the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests `AVG(X)` and a materialized view contains `SUM(X)` and `COUNT(X)`, then `AVG(X)` can be computed as `SUM(X)/COUNT(X)`.

If the grouping compatibility check determined that the rollup of aggregates stored in a materialized view is required, then the aggregate computability check determines if it is possible to roll up each aggregate requested by the query using aggregates in the materialized view.

Query Rewrite Using Dimensions

This section discusses the following aspects of using dimensions in a rewrite environment:

- Benefits of Using Dimensions
- How to Define Dimensions

Benefits of Using Dimensions

A dimension defines a hierarchical (parent/child) relationships between columns, where all the columns do not have to come from the same table.

Dimension definitions increase the possibility of query rewrite because they help to establish functional dependencies between the columns. In addition, dimensions can express intra-table relationships that cannot be expressed by constraints. A dimension definition does not occupy additional storage. Rather, a dimension definition establishes metadata that describes the intra- and inter-dimensional relationships within your schema. Before creating a materialized view, the first step is to review the schema and define the dimensions as this can significantly improve the chances of rewriting a query.

How to Define Dimensions

For any given schema, dimensions can be created by following the following steps.

Step 1 Identify all dimensions and dimension tables in the schema.

If the dimensions are normalized, that is, stored in multiple tables, then check that a join between the dimension tables guarantees that each child-side row joins with one and only one parent-side row. In the case of denormalized dimensions, check that the child-side columns uniquely determine the parent-side (or attribute) columns. Failure to abide by these rules may result in incorrect results being returned from queries.

Step 2 Identify the hierarchies within each dimension.

As an example, day is a child of month (we can aggregate day level data up to month), and quarter is a child of year.

Step 3 Identify the attribute dependencies within each level of the hierarchy.

As an example, identify that `calendar_month_name` is an attribute of month.

Step 4 Identify joins from the fact table in a data warehouse to each dimension.

Then check that each join can guarantee that each fact row joins with one and only one dimension row. This condition must be declared, and optionally enforced, by adding `FOREIGN KEY` and `NOT NULL` constraints on the fact key columns and `PRIMARY KEY` constraints on the parent-side join keys. If these relationships can be guaranteed by other data handling procedures (for example, your load process), these constraints can be enabled using the `NOVALIDATE` option to avoid the time required to validate that every row in the table conforms to the constraints. The `RELY` clause is also required for all nonvalidated constraints to make them eligible for use in query rewrite.

Example SQL Statement to Create Time Dimension

```
CREATE DIMENSION times_dim
LEVEL day IS TIMES.TIME_ID
LEVEL month IS TIMES.CALENDAR_MONTH_DESC
LEVEL quarter IS TIMES.CALENDAR_QUARTER_DESC
LEVEL year IS TIMES.CALENDAR_YEAR
LEVEL fis_week IS TIMES.WEEK_ENDING_DAY
LEVEL fis_month  IS TIMES.FISCAL_MONTH_DESC
LEVEL fis_quarter IS TIMES.FISCAL_QUARTER_DESC
LEVEL fis_year IS TIMES.FISCAL_YEAR
        HIERARCHY cal_rollup
        (day  CHILD OF month CHILD OF quarter CHILD OF year)
        HIERARCHY fis_rollup
        (day  CHILD OF fis_week CHILD OF fis_month CHILD OF fis_quarter
        CHILD OF fis_year)

        ATTRIBUTE day DETERMINES
        (day_number_in_week, day_name, day_number_in_month,
         calendar_week_number)

        ATTRIBUTE month DETERMINES
        (calendar_month_desc, calendar_month_number, calendar_month_name,
         days_in_cal_month, end_of_cal_month)

        ATTRIBUTE quarter DETERMINES
        (calendar_quarter_desc, calendar_quarter_number,days_in_cal_quarter,
         end_of_cal_quarter)

        ATTRIBUTE year DETERMINES
```

```
(calendar_year,  days_in_cal_year, end_of_cal_year)

ATTRIBUTE fis_week DETERMINES
(week_ending_day, fiscal_week_number);
```

Remember to set the parameter `QUERY_REWRITE_INTEGRITY` to `TRUSTED` or `STALE_TOLERATED` for query rewrite to take advantage of the relationships declared in dimensions.

Types of Query Rewrite

Queries that have aggregates that require computations over a large number of rows or joins between very large tables can be expensive and thus can take a long time to return the results. Query rewrite transparently rewrites such queries using materialized views that have pre-computed results, so that the queries can be answered almost instantaneously. These materialized views can be broadly categorized into two groups, namely materialized aggregate views and materialized join views. Materialized aggregate views are tables that have pre-computed aggregate values for columns from original tables. Similarly, materialized join views are tables that have pre-computed joins between columns from original tables. Query rewrite transforms an incoming query to fetch the results from materialized view columns. Because these columns contain already pre-computed results, the incoming query can be answered almost instantaneously. For considerations regarding query rewrite of cube organized materialized views, see *Oracle OLAP User's Guide*.

This section discusses the following methods that can be used to rewrite a query:

- Text Match Rewrite
- Join Back
- Aggregate Computability
- Aggregate Rollup
- Rollup Using a Dimension
- When Materialized Views Have Only a Subset of Data
- Partition Change Tracking (PCT) Rewrite
- Multiple Materialized Views

Text Match Rewrite

The query rewrite engine always initially tries to compare the text of incoming query with the text of the definition of any potential materialized views to rewrite the query. This is because the overhead of doing a simple text comparison is usually negligible comparing to the cost of doing a complex analysis required for the general rewrite.

The query rewrite engine uses two text match methods, full text match rewrite and partial text match rewrite. In full text match the entire text of a query is compared against the entire text of a materialized view definition (that is, the entire `SELECT` expression), ignoring the white space during text comparison. For example, assume that we have the following materialized view, `sum_sales_pscat_month_city_mv`:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_month_city_mv
ENABLE QUERY REWRITE AS
   SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
   SUM(s.amount_sold) AS sum_amount_sold,
   COUNT(s.amount_sold) AS count_amount_sold
```

```
   FROM sales s, products p, times t, customers c
   WHERE s.time_id=t.time_id
      AND     s.prod_id=p.prod_id
      AND     s.cust_id=c.cust_id
  GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

Consider the following query:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold,
       COUNT(s.amount_sold) AS count_amount_sold
       FROM sales s, products p, times t, customers c
       WHERE s.time_id=t.time_id
         AND     s.prod_id=p.prod_id
         AND     s.cust_id=c.cust_id
       GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

This query matches `sum_sales_pscat_month_city_mv` (white space excluded) and is rewritten as:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
       mv.sum_amount_sold, mv.count_amount_sold
FROM   sum_sales_pscat_month_city_mv;
```

When full text match fails, the optimizer then attempts a partial text match. In this method, the text starting from the `FROM` clause of a query is compared against the text starting with the `FROM` clause of a materialized view definition. Therefore, the following query can be rewritten:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
       AVG(s.amount_sold)
FROM   sales s, products p, times t, customers c
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
AND    s.cust_id=c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc, c.cust_city;
```

This query is rewritten as:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
       mv.sum_amount_sold/mv.count_amount_sold
FROM   sum_sales_pscat_month_city_mv mv;
```

Note that, under the partial text match rewrite method, the average of sales aggregate required by the query is computed using the sum of sales and count of sales aggregates stored in the materialized view.

When neither text match succeeds, the optimizer uses a general query rewrite method.

Text match rewrite can distinguish contexts where the difference between uppercase and lowercase is significant and where it is not. For example, the following statements are equivalent:

```
SELECT X, 'aBc' FROM Y

Select x, 'aBc' From y
```

Join Back

If some column data requested by a query cannot be obtained from a materialized view, the optimizer further determines if it can be obtained based on a data relationship called a functional dependency. When the data in a column can determine

data in another column, such a relationship is called a functional dependency or functional determinance. For example, if a table contains a primary key column called prod_id and another column called prod_name, then, given a prod_id value, it is possible to look up the corresponding prod_name. The opposite is not true, which means a prod_name value need not relate to a unique prod_id.

When the column data required by a query is not available from a materialized view, such column data can still be obtained by joining the materialized view back to the table that contains required column data provided the materialized view contains a key that functionally determines the required column data. For example, consider the following query:

```
SELECT p.prod_category, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id  AND s.prod_id=p.prod_id AND p.prod_category='CD'
GROUP BY p.prod_category, t.week_ending_day;
```

The materialized view sum_sales_prod_week_mv contains p.prod_id, but not p.prod_category. However, you can join sum_sales_prod_week_mv back to products to retrieve prod_category because prod_id functionally determines prod_category. The optimizer rewrites this query using sum_sales_prod_week_mv as follows:

```
SELECT p.prod_category, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_prod_week_mv mv, products p
WHERE  mv.prod_id=p.prod_id AND p.prod_category='CD'
GROUP BY p.prod_category, mv.week_ending_day;
```

Here the products table is called a joinback table because it was originally joined in the materialized view but joined again in the rewritten query.

You can declare functional dependency in two ways:

- Using the primary key constraint (as shown in the previous example)
- Using the DETERMINES clause of a dimension

The DETERMINES clause of a dimension definition might be the only way you could declare functional dependency when the column that determines another column cannot be a primary key. For example, the products table is a denormalized dimension table that has columns prod_id, prod_name, and prod_subcategory that functionally determines prod_subcat_desc and prod_category that determines prod_cat_desc.

The first functional dependency can be established by declaring prod_id as the primary key, but not the second functional dependency because the prod_subcategory column contains duplicate values. In this situation, you can use the DETERMINES clause of a dimension to declare the second functional dependency.

The following dimension definition illustrates how functional dependencies are declared:

```
CREATE DIMENSION products_dim
        LEVEL product           IS (products.prod_id)
        LEVEL subcategory       IS (products.prod_subcategory)
        LEVEL category          IS (products.prod_category)
        HIERARCHY prod_rollup (
                product         CHILD OF
                subcategory     CHILD OF
                category
        )
        ATTRIBUTE product DETERMINES products.prod_name
```

```
        ATTRIBUTE product DETERMINES products.prod_desc
        ATTRIBUTE subcategory DETERMINES products.prod_subcat_desc
        ATTRIBUTE category DETERMINES products.prod_cat_desc;
```

The hierarchy `prod_rollup` declares hierarchical relationships that are also `1:n` functional dependencies. The `1:1` functional dependencies are declared using the `DETERMINES` clause, as seen when `prod_subcategory` functionally determines `prod_subcat_desc`.

If the following materialized view is created:

```
CREATE MATERIALIZED VIEW sum_sales_pscat_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_subcategory, t.week_ending_day,
       SUM(s.amount_sold) AS sum_amount_sole
FROM sales s, products p, times t
WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id
GROUP BY p.prod_subcategory, t.week_ending_day;
```

Then consider the following query:

```
SELECT p.prod_subcategory_desc, t.week_ending_day, SUM(s.amount_sold)
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
AND    p.prod_subcat_desc LIKE '%Men'
GROUP BY p.prod_subcat_desc, t.week_ending_day;
```

This can be rewritten by joining `sum_sales_pscat_week_mv` to the `products` table so that `prod_subcat_desc` is available to evaluate the predicate. However, the join is based on the `prod_subcategory` column, which is not a primary key in the `products` table; therefore, it allows duplicates. This is accomplished by using an inline view that selects distinct values and this view is joined to the materialized view as shown in the rewritten query.

```
SELECT iv.prod_subcat_desc, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM  sum_sales_pscat_week_mv mv,
     (SELECT DISTINCT prod_subcategory, prod_subcat_desc
      FROM products) iv
WHERE  mv.prod_subcategory=iv.prod_subcategory
AND iv.prod_subcat_desc LIKE '%Men'
GROUP BY iv.prod_subcat_desc, mv.week_ending_day;
```

This type of rewrite is possible because `prod_subcategory` functionally determines `prod_subcategory_desc` as declared in the dimension.

Aggregate Computability

Query rewrite can also occur when the optimizer determines if the aggregates requested by a query can be derived or computed from one or more aggregates stored in a materialized view. For example, if a query requests `AVG(X)` and a materialized view contains `SUM(X)` and `COUNT(X)`, then `AVG(X)` can be computed as `SUM(X)/COUNT(X)`.

In addition, if it is determined that the rollup of aggregates stored in a materialized view is required, then, if it is possible, query rewrite also rolls up each aggregate requested by the query using aggregates in the materialized view.

For example, `SUM(sales)` at the city level can be rolled up to `SUM(sales)` at the state level by summing all `SUM(sales)` aggregates in a group with the same state value. However, `AVG(sales)` cannot be rolled up to a coarser level unless

COUNT(sales) or SUM(sales) is also available in the materialized view. Similarly, VARIANCE(sales) or STDDEV(sales) cannot be rolled up unless both COUNT(sales) and SUM(sales) are also available in the materialized view. For example, consider the following query:

```
ALTER TABLE times MODIFY CONSTRAINT time_pk RELY;
ALTER TABLE customers MODIFY CONSTRAINT customers_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_time_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_customer_fk RELY;
SELECT  p.prod_subcategory, AVG(s.amount_sold) AS avg_sales
FROM  sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

This statement can be rewritten with materialized view sum_sales_pscat_month_city_mv provided the join between sales and times and sales and customers are lossless and non-duplicating. Further, the query groups by prod_subcategory whereas the materialized view groups by prod_subcategory, calendar_month_desc and cust_city, which means the aggregates stored in the materialized view have to be rolled up. The optimizer rewrites the query as the following:

```
SELECT mv.prod_subcategory, SUM(mv.sum_amount_sold)/COUNT(mv.count_amount_sold)
   AS avg_sales
FROM sum_sales_pscat_month_city_mv mv
GROUP BY mv.prod_subcategory;
```

The argument of an aggregate such as SUM can be an arithmetic expression such as A+B. The optimizer tries to match an aggregate SUM(A+B) in a query with an aggregate SUM(A+B) or SUM(B+A) stored in a materialized view. In other words, expression equivalence is used when matching the argument of an aggregate in a query with the argument of a similar aggregate in a materialized view. To accomplish this, Oracle converts the aggregate argument expression into a canonical form such that two different but equivalent expressions convert into the same canonical form. For example, A*(B-C), A*B-C*A, (B-C)*A, and -A*C+A*B all convert into the same canonical form and, therefore, they are successfully matched.

Aggregate Rollup

If the grouping of data requested by a query is at a coarser level than the grouping of data stored in a materialized view, the optimizer can still use the materialized view to rewrite the query. For example, the materialized view sum_sales_pscat_week_mv groups by prod_subcategory and week_ending_day. This query groups by prod_subcategory, a coarser grouping granularity:

```
ALTER TABLE times MODIFY CONSTRAINT time_pk RELY;
ALTER TABLE sales MODIFY CONSTRAINT sales_time_fk RELY;
SELECT p.prod_subcategory, SUM(s.amount_sold) AS sum_amount
FROM   sales s, products p
WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

Therefore, the optimizer rewrites this query as:

```
SELECT mv.prod_subcategory, SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv
GROUP BY mv.prod_subcategory;
```

Rollup Using a Dimension

When reporting is required at different levels in a hierarchy, materialized views do not have to be created at each level in the hierarchy provided dimensions have been defined. This is because query rewrite can use the relationship information in the dimension to roll up the data in the materialized view to the required level in the hierarchy.

In the following example, a query requests data grouped by `prod_category` while a materialized view stores data grouped by `prod_subcategory`. If `prod_subcategory` is a `CHILD OF` `prod_category` (see the dimension example earlier), the grouped data stored in the materialized view can be further grouped by `prod_category` when the query is rewritten. In other words, aggregates at `prod_subcategory` level (finer granularity) stored in a materialized view can be rolled up into aggregates at `prod_category` level (coarser granularity).

For example, consider the following query:

```
SELECT p.prod_category, t.week_ending_day, SUM(s.amount_sold) AS sum_amount
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_category, t.week_ending_day;
```

Because `prod_subcategory` functionally determines `prod_category`, `sum_sales_pscat_week_mv` can be used with a joinback to `products` to retrieve `prod_category` column data, and then aggregates can be rolled up to `prod_category` level, as shown in the following:

```
SELECT pv.prod_category, mv.week_ending_day, SUM(mv.sum_amount_sold)
FROM   sum_sales_pscat_week_mv mv,
       (SELECT DISTINCT prod_subcategory, prod_category
        FROM products) pv
WHERE mv.prod_subcategory= pv.prod_subcategory
GROUP BY pv.prod_category, mv.week_ending_day;
```

When Materialized Views Have Only a Subset of Data

Oracle supports rewriting of queries so that they will use materialized views in which the `HAVING` or `WHERE` clause of the materialized view contains a selection of a subset of the data in a table or tables. For example, only those customers who live in New Hampshire. In other words, the `WHERE` clause in the materialized view will be `WHERE state = 'New Hampshire'`.

To perform this type of query rewrite, Oracle must determine if the data requested in the query is contained in, or is a subset of, the data stored in the materialized view. The following sections detail the conditions where Oracle can solve this problem and thus rewrite a query to use a materialized view that contains a filtered portion of the data in the detail table.

To determine if query rewrite can occur on filtered data, a selection computability check is performed when both the query and the materialized view contain selections (non-joins) and the check is done on the `WHERE` as well as the `HAVING` clause. If the materialized view contains selections and the query does not, then the selection compatibility check fails because the materialized view is more restrictive than the query. If the query has selections and the materialized view does not, then the selection compatibility check is not needed.

A materialized view's `WHERE` or `HAVING` clause can contain a join, a selection, or both, and still be used to rewrite a query. Predicate clauses containing expressions, or

selecting rows based on the values of particular columns, are examples of non-join predicates.

Query Rewrite Definitions

Before describing what is possible when query rewrite works with only a subset of the data, the following definitions are useful:

- *join relop*

  Is one of the following (=, <, <=, >, >=)

- *selection relop*

  Is one of the following (=, <, <=, >, >=, !=, [NOT] BETWEEN | IN| LIKE |NULL)

- *join predicate*

  Is of the form (*column1 join relop column2*), where columns are from different tables within the same FROM clause in the current query block. So, for example, an outer reference is not possible.

- *selection predicate*

  Is of the form *left-hand-side-expression relop right-hand-side-expression*. All non-join predicates are selection predicates. The left-hand side usually contains a column and the right-hand side contains the values. For example, color='red' means the left-hand side is color and the right-hand side is 'red' and the relational operator is (=).

Selection Categories

Selections are categorized into the following cases:

- Simple

  Simple selections are of the form *expression relop constant*.

- Complex

  Complex selections are of the form *expression relop expression*.

- Range

  Range selections are of a form such as WHERE (cust_last_name BETWEEN 'abacrombe' AND 'anakin').

  Note that simple selections with relational operators (<,<=,>,>=) are also considered range selections.

- IN-lists

  Single and multi-column IN-lists such as WHERE(prod_id) IN (102, 233, ...).

  Note that selections of the form (column1='v1' OR column1='v2' OR column1='v3' OR ....) are treated as a group and classified as an IN-list.

- IS [NOT] NULL

- [NOT] LIKE

- Other

  Other selections are when it cannot determine the boundaries for the data. For example, EXISTS.

When comparing a selection from the query with a selection from the materialized view, the left-hand side of both selections are compared.

If the left-hand side selections match, then the right-hand side values are checked for containment. That is, the right-hand side values of the query selection must be contained by right-hand side values of the materialized view selection.

You can also use expressions in selection predicates. This process resembles the following:

```
expression relational operator constant
```

Where *expression* can be any arbitrary arithmetic expression allowed by the Oracle Database. The expression in the materialized view and the query must match. Oracle attempts to discern expressions that are logically equivalent, such as `A+B` and `B+A`, and always recognizes identical expressions as being equivalent.

You can also use queries with an expression on both sides of the operator or user-defined functions as operators. Query rewrite occurs when the complex predicate in the materialized view and the query are logically equivalent. This means that, unlike exact text match, terms could be in a different order and rewrite can still occur, as long as the expressions are equivalent.

Examples of Query Rewrite Selection

Here are a number of examples showing how query rewrite can still occur when the data is being filtered.

Example 19–1 Single Value Selection

If the query contains the following clause:

```
WHERE prod_id = 102
```

And, if a materialized view contains the following clause:

```
WHERE prod_id BETWEEN 0 AND 200
```

Then, the left-hand side selections match on `prod_id` and the right-hand side value of the query `102` is within the range of the materialized view, so query rewrite is possible.

Example 19–2 Bounded Range Selection

A selection can be a bounded range (a range with an upper and lower value). For example, if the query contains the following clause:

```
WHERE prod_id > 10 AND prod_id < 50
```

And if a materialized view contains the following clause:

```
WHERE prod_id BETWEEN 0 AND 200
```

Then, the selections are matched on `prod_id` and the query range is within the materialized view range. In this example, notice that both query selections are based on the same column.

Example 19–3 Selection With Expression

If the query contains the following clause:

```
WHERE (sales.amount_sold * .07) BETWEEN 1.00 AND 100.00
```

And if a materialized view contains the following clause:

```
WHERE (sales.amount_sold * .07) BETWEEN 0.0 AND 200.00
```

Then, the selections are matched on `(sales.amount_sold *.07)` and the right-hand side value of the query is within the range of the materialized view, therefore query rewrite is possible. Complex selections such as this require that the left-hand side and the right-hand side be matched within range of the materialized view.

Example 19–4 Exact Match Selections

If the query contains the following clause:

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

And if a materialized view contains the following:

```
WHERE (cost.unit_price * 0.95) > (cost_unit_cost  * 1.25)
```

If the left-hand side and the right-hand side match the materialized view and the *selection_relop* is the same, then the selection can usually be dropped from the rewritten query. Otherwise, the selection must be kept to filter out extra data from the materialized view.

If query rewrite can drop the selection from the rewritten query, all columns from the selection may not have to be in the materialized view so more rewrites can be done. This ensures that the materialized view data is not more restrictive than the query.

Example 19–5 More Selection in the Query

Selections in the query do not have to be matched by any selections in the materialized view but, if they are, then the right-hand side values must be contained by the materialized view. For example, if the query contains the following clause:

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

And if a materialized view contains the following clause:

```
WHERE prod_category = 'Men'
```

Then, in this example, only selection with `prod_category` is matched. The query has an extra selection that is not matched but this is acceptable because if the materialized view selects `prod_name` or selects a column that can be joined back to the detail table to get `prod_name`, then query rewrite is possible. The only requirement is that query rewrite must have a way of applying the `prod_name` selection to the materialized view.

Example 19–6 No Rewrite Because of Fewer Selections in the Query

If the query contains the following clause:

```
WHERE prod_category = 'Men'
```

And if a materialized view contains the following clause:

```
WHERE prod_name = 'Shorts' AND prod_category = 'Men'
```

Then, the materialized view selection with `prod_name` is not matched. The materialized view is more restrictive that the query because it only contains the product Shorts, therefore, query rewrite does not occur.

Example 19–7 Multi-Column IN-List Selections

Query rewrite also checks for cases where the query has a multi-column IN-list where the columns are fully matched by individual columns from the materialized view single column IN-lists. For example, if the query contains the following:

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1033, 2000))
```

And if a materialized view contains the following:

```
WHERE prod_id IN (1022,1033) AND cust_id IN (1000, 2000)
```

Then, the materialized view IN-lists are matched by the columns in the query multi-column IN-list. Furthermore, the right-hand side values of the query selection are contained by the materialized view so that rewrite occurs.

Example 19–8 Selections Using IN-Lists

Selection compatibility also checks for cases where the materialized view has a multi-column IN-list where the columns are fully matched by individual columns or columns from IN-lists in the query. For example, if the query contains the following:

```
WHERE prod_id = 1022 AND cust_id IN (1000, 2000)
```

And if a materialized view contains the following:

```
WHERE (prod_id, cust_id) IN ((1022, 1000), (1022, 2000))
```

Then, the materialized view IN-list columns are fully matched by the columns in the query selections. Furthermore, the right-hand side values of the query selection are contained by the materialized view. So rewrite succeeds.

Example 19–9 Multiple Selections or Expressions

If the query contains the following clause:

```
WHERE (city_population > 15000 AND city_population < 25000
   AND state_name = 'New Hampshire')
```

And if a materialized view contains the following clause:

```
WHERE (city_population < 5000 AND state_name = 'New York') OR
   (city_population BETWEEN 10000 AND 50000 AND state_name = 'New Hampshire')
```

Then, the query is said to have a single disjunct (group of selections separated by AND) and the materialized view has two disjuncts separated by OR. The single query disjunct is contained by the second materialized view disjunct so selection compatibility succeeds. It is clear that the materialized view contains more data than needed by the query so the query can be rewritten.

Handling of the HAVING Clause in Query Rewrite

Query rewrite can also occur when the query specifies a range of values for an aggregate in the HAVING clause, such as SUM(s.amount_sold) BETWEEN 10000 AND 20000, as long as the range specified is within the range specified in the materialized view.

```
CREATE MATERIALIZED VIEW product_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
```

```
FROM products p, sales s
WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 5000 AND 50000;
```

Then, a query such as the following could be rewritten:

```
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM products p, sales s WHERE p.prod_id = s.prod_id
GROUP BY prod_name
HAVING SUM(s.amount_sold) BETWEEN 10000 AND 20000;
```

This query is rewritten as follows:

```
SELECT mv.prod_name, mv.dollar_sales FROM product_sales_mv mv
WHERE mv.dollar_sales BETWEEN 10000 AND 20000;
```

Query Rewrite When the Materialized View has an IN-List

You can use query rewrite when the materialized view contains an `IN`-list. For example, given the following materialized view definition:

```
CREATE MATERIALIZED VIEW popular_promo_sales_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT p.promo_name, SUM(s.amount_sold) AS sum_amount_sold
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id
AND p.promo_name IN ('coupon', 'premium', 'giveaway')
GROUP BY promo_name;
```

The following query can be rewritten:

```
SELECT p.promo_name, SUM(s.amount_sold)
FROM  promotions p, sales s
WHERE s.promo_id = p.promo_id AND p.promo_name IN ('coupon', 'premium')
GROUP BY p.promo_name;
```

This query is rewritten as follows:

```
SELECT * FROM popular_promo_sales_mv mv
WHERE mv.promo_name IN ('coupon', 'premium');
```

Partition Change Tracking (PCT) Rewrite

PCT rewrite enables the optimizer to accurately rewrite queries with fresh data using materialized views that are only partially fresh. To do so, Oracle Database keeps track of which partitions in the detail tables have been updated. Oracle Database then tracks which rows in the materialized view originate from the affected partitions in the detail tables. The optimizer is then able to use those portions of the materialized view that are known to be fresh. You can check details about freshness with the `DBA_MVIEWS`, `DBA_DETAIL_RELATIONS`, and `DBA_MVIEW_DETAIL_PARTITION` views. See "Viewing Partition Freshness" on page 16-16 for examples of using these views.

The optimizer uses PCT rewrite in `QUERY_REWRITE_INTEGRITY = ENFORCED` and `TRUSTED` modes. The optimizer does not use PCT rewrite in `STALE_TOLERATED` mode because data freshness is not considered in that mode. Also, for PCT rewrite to occur, a `WHERE` clause is required.

You can use PCT rewrite with partitioning, but hash partitioning is not supported. The following sections discuss aspects of using PCT:

- PCT Rewrite Based on Range Partitioned Tables
- PCT Rewrite Based on Range-List Partitioned Tables
- PCT Rewrite Based on List Partitioned Tables
- PCT Rewrite and PMARKER
- PCT Rewrite Using Rowid as PMARKER

PCT Rewrite Based on Range Partitioned Tables

The following example illustrates a PCT rewrite example where the materialized view is PCT enabled through partition key and the underlying base table is range partitioned on the time key.

```
CREATE TABLE part_sales_by_time (time_id, prod_id, amount_sold,
       quantity_sold)
  PARTITION BY RANGE (time_id)
  (
    PARTITION old_data
      VALUES LESS THAN (TO_DATE('01-01-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter1
      VALUES LESS THAN (TO_DATE('01-04-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter2
      VALUES LESS THAN (TO_DATE('01-07-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter3
      VALUES LESS THAN (TO_DATE('01-10-1999', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION quarter4
      VALUES LESS THAN (TO_DATE('01-01-2000', 'DD-MM-YYYY'))
      PCTFREE 0
      STORAGE (INITIAL 8M),
    PARTITION max_partition
      VALUES LESS THAN (MAXVALUE)
      PCTFREE 0
      STORAGE (INITIAL 8M)
  )
  AS
  SELECT s.time_id, s.prod_id, s.amount_sold, s.quantity_sold
  FROM sales s;
```

Then create a materialized view that contains the total number of products sold by date.

```
CREATE MATERIALIZED VIEW  sales_in_1999_mv
  BUILD IMMEDIATE
  REFRESH FORCE ON DEMAND
  ENABLE QUERY REWRITE
  AS
  SELECT s.time_id, s.prod_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
    AND s.time_id BETWEEN TO_DATE('01-01-1999', 'DD-MM-YYYY')
    AND TO_DATE('31-12-1999', 'DD-MM-YYYY')
```

```
  GROUP BY s.time_id, s.prod_id, p.prod_name;
```

Note that the following query will be rewritten with materialized view sales_in_1999_mv:

```
SELECT s.time_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
    AND s.time_id < TO_DATE(''01-02-1999'', ''DD-MM-YYYY'')
    AND s.time_id >= TO_DATE(''01-01-1999'', ''DD-MM-YYYY'')
  GROUP BY s.time_id, p.prod_name');
```

If we add a row to quarter4 in part_sales_by_time as:

```
INSERT INTO part_sales_by_time
  VALUES (TO_DATE('26-12-1999', 'DD-MM-YYYY'),38920,2500, 20);

commit;
```

Then the materialized view sales_in_1999_mv becomes stale. With PCT rewrite, we can rewrite queries that request data from only the fresh portions of the materialized view. Note that since the materialized view sales_in_1999_mv has the time_id in its SELECT and GROUP BY clause, it is PCT enabled so the following query will be rewritten successfully as no data from quarter4 is requested.

```
SELECT s.time_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
  AND s.time_id < TO_DATE(''01-07-1999'', ''DD-MM-YYYY'')
  AND s.time_id >= TO_DATE(''01-03-1999'', ''DD-MM-YYYY'')
  GROUP BY s.time_id, p.prod_name');
```

The following query cannot be rewritten if multiple materialized view rewrite is set to off. Because multiple materialized view rewrite is on by default, the following query is rewritten with materialized view and base tables:

```
SELECT s.time_id, p.prod_name, SUM(quantity_sold)
  FROM part_sales_by_time s, products p
  WHERE p.prod_id = s.prod_id
  AND s.time_id < TO_DATE(''31-10-1999'', ''DD-MM-YYYY'') AND
       s.time_id > TO_DATE(''01-07-1999'', ''DD-MM-YYYY'')
  GROUP BY s.time_id, p.prod_name');
```

PCT Rewrite Based on Range-List Partitioned Tables

If the detail table is range-list partitioned, a materialized view that depends on this detail table can support PCT at both the partitioning and subpartitioning levels. If both the partition and subpartition keys are present in the materialized view, PCT can be done at a finer granularity; materialized view refreshes can be done to smaller portions of the materialized view and more queries could be rewritten with a stale materialized view. Alternatively, if only the partition key is present in the materialized view, PCT can be done with courser granularity.

Consider the following range-list partitioned table:

```
CREATE TABLE sales_par_range_list
 (calendar_year, calendar_month_number, day_number_in_month,
  country_name, prod_id, prod_name, quantity_sold, amount_sold)
PARTITION BY RANGE (calendar_month_number)
SUBPARTITION BY LIST (country_name)
 (PARTITION q1 VALUES LESS THAN (4)
```

```
 (SUBPARTITION q1_America VALUES
 ('United States of America', 'Argentina'),
   SUBPARTITION q1_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q1_Europe VALUES ('France', 'Spain', 'Ireland')),
   PARTITION q2 VALUES LESS THAN (7)
  (SUBPARTITION q2_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q2_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q2_Europe VALUES ('France', 'Spain', 'Ireland')),
     PARTITION q3 VALUES LESS THAN (10)
  (SUBPARTITION q3_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q3_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q3_Europe VALUES ('France', 'Spain', 'Ireland')),
     PARTITION q4 VALUES LESS THAN (13)
  (SUBPARTITION q4_America VALUES
   ('United States of America', 'Argentina'),
   SUBPARTITION q4_Asia VALUES ('Japan', 'India'),
   SUBPARTITION q4_Europe VALUES ('France', 'Spain', 'Ireland')))
  AS SELECT t.calendar_year, t.calendar_month_number,
     t.day_number_in_month, c1.country_name, s.prod_id,
     p.prod_name, s.quantity_sold, s.amount_sold
  FROM times t, countries c1, products p, sales s, customers c2
  WHERE s.time_id = t.time_id AND s.prod_id = p.prod_id AND
        s.cust_id = c2.cust_id AND c2.country_id = c1.country_id AND
        c1.country_name IN ('United States of America', 'Argentina',
          'Japan', 'India', 'France', 'Spain', 'Ireland');
```

Then consider the following materialized view `sum_sales_per_year_month_mv`, which has the total amount of products sold each month of each year:

```
CREATE MATERIALIZED VIEW  sum_sales_per_year_month_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year, s.calendar_month_number,
         SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s WHERE s.calendar_year > 1990
GROUP BY s.calendar_year, s.calendar_month_number;
```

`sales_per_country_mv` supports PCT against `sales_par_range_list` at the range partitioning level as its range partition key `calendar_month_number` is in its `SELECT` and `GROUP BY` list:

```
INSERT INTO sales_par_range_list
   VALUES (2001, 3, 25, 'Spain', 20, 'PROD20',  300,  20.50);
```

This statement inserts a row with `calendar_month_number = 3` and `country_name = 'Spain'`. This row is inserted into partition `q1` subpartition `Europe`. After this `INSERT` statement, `sum_sales_per_year_month_mv` is stale with respect to partition `q1` of `sales_par_range_list`. So any incoming query that accesses data from this partition in `sales_par_range_list` cannot be rewritten, for example, the following statement:

Note that the following query accesses data from partitions `q1` and `q2`. Because q1 was updated, the materialized view is stale with respect to `q1` so PCT rewrite is unavailable.

```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000
```

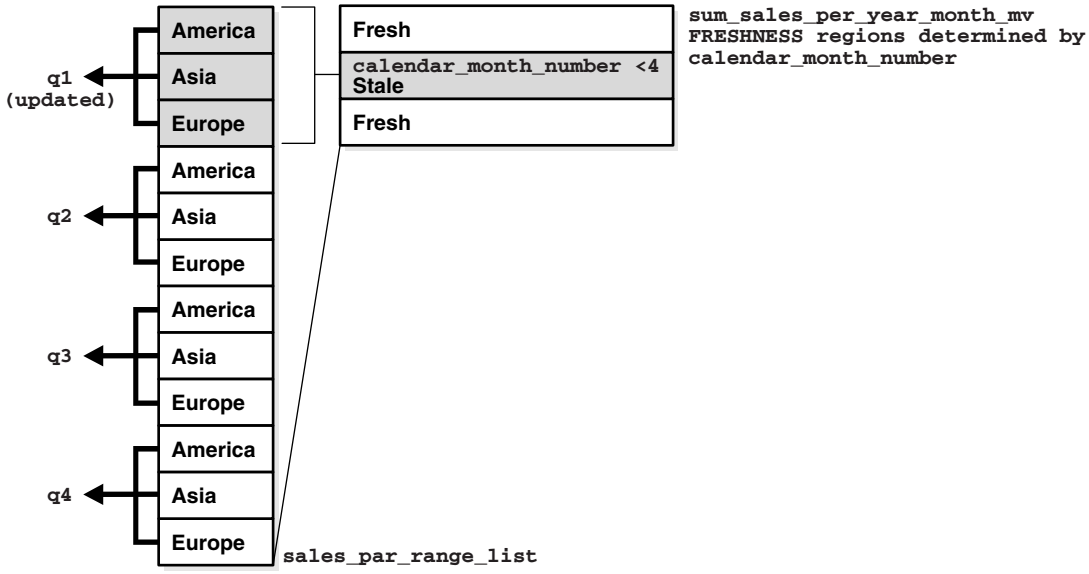
```
  AND s.calendar_month_number BETWEEN 2 AND 6
GROUP BY s.calendar_year;
```

An example of a statement that does rewrite after the INSERT statement is the following, because it accesses fresh material:

```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000 AND s.calendar_month_number BETWEEN 5 AND 9
GROUP BY s.calendar_year;
```

Figure 19–3 offers a graphical illustration of what is stale and what is fresh.

Figure 19–3 PCT Rewrite and Range-List Partitioning



PCT Rewrite Based on List Partitioned Tables

If the LIST partitioning key is present in the materialized view's SELECT and GROUP BY, then PCT will be supported by the materialized view. Regardless of the supported partitioning type, if the partition marker or rowid of the detail table is present in the materialized view then PCT is supported by the materialized view on that specific detail table.

```
CREATE TABLE sales_par_list
(calendar_year, calendar_month_number, day_number_in_month,
 country_name, prod_id, quantity_sold, amount_sold)
 PARTITION BY LIST (country_name)
 (PARTITION America
      VALUES ('United States of America', 'Argentina'),
  PARTITION Asia
      VALUES ('Japan', 'India'),
  PARTITION Europe
      VALUES ('France', 'Spain', 'Ireland'))
  AS SELECT t.calendar_year, t.calendar_month_number,
         t.day_number_in_month, c1.country_name, s.prod_id,
         s.quantity_sold, s.amount_sold
  FROM times t, countries c1, sales s, customers c2
  WHERE s.time_id = t.time_id and s.cust_id = c2.cust_id and
```

```
 c2.country_id = c1.country_id and
 c1.country_name IN ('United States of America', 'Argentina',
'Japan', 'India', 'France', 'Spain', 'Ireland');
```

If a materialized view is created on the table sales_par_list, which has a list partitioning key, PCT rewrite will use that materialized view for potential rewrites.

To illustrate this feature, the following example creates a materialized view that has the total amounts sold of every product in each country for each year. The view depends on detail tables sales_par_list and products.

```
CREATE MATERIALIZED VIEW sales_per_country_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year AS calendar_year, s.country_name AS country_name,
 p.prod_name AS prod_name, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year <= 2000
GROUP BY s.calendar_year, s.country_name, prod_name;
```

sales_per_country_mv supports PCT against sales_par_list as its list partition key country_name is in its SELECT and GROUP BY list. Table products is not partitioned, so sales_per_country_mv does not support PCT against this table.

A query could be rewritten (in ENFORCED or TRUSTED modes) in terms of sales_per_country_mv even if sales_per_country_mv is stale if the incoming query accesses only fresh parts of the materialized view. You can determine which parts of the materialized view are FRESH only if the updated tables are PCT enabled in the materialized view. If non-PCT enabled tables have been updated, then the rewrite is not possible with fresh data from that specific materialized view as you cannot identify the FRESH portions of the materialized view.

sales_per_country_mv supports PCT on sales_par_list and does not support PCT on table product. If table products is updated, then PCT rewrite is not possible with sales_per_country_mv as you cannot tell which portions of the materialized view are FRESH.

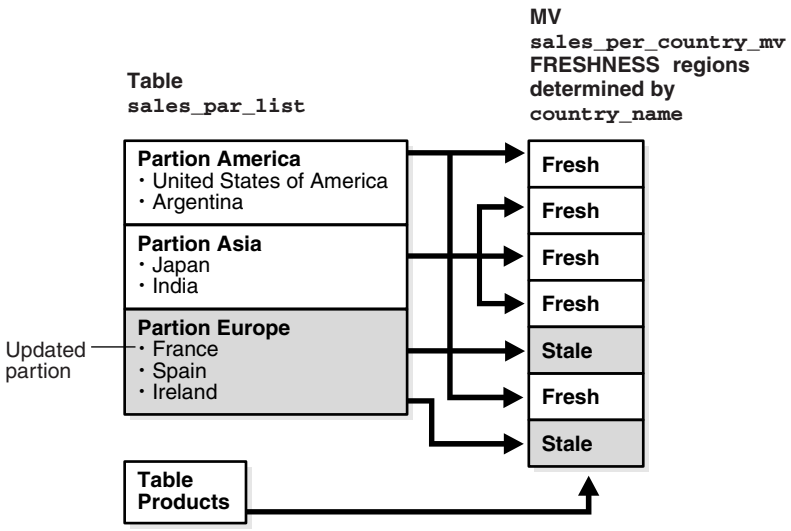
The following updates sales_par_list as follows:

```
INSERT INTO sales_par_list VALUES (2000, 10, 22, 'France', 900, 20, 200.99);
```

This statement inserted a row into partition Europe in table sales_par_list. Now sales_per_country_mv is stale, but PCT rewrite (in ENFORCED and TRUSTED modes) is possible as this materialized view supports PCT against table sales_par_list. The fresh and stale areas of the materialized view are identified based on the partitioned detail table sales_par_list.

Figure 19–4 illustrates what is fresh and what is stale in this example.

Figure 19–4 PCT Rewrite and List Partitioning



Consider the following query:

```
SELECT s.country_name, p.prod_name, SUM(s.amount_sold) AS sum_sales,
   COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year = 2000
  AND s.country_name IN ('United States of America', 'Japan')
GROUP BY s.country_name, p.prod_name;
```

This query accesses partitions `America` and `Asia` in `sales_par_list`; these partition have not been updated so rewrite is possible with stale materialized view `sales_per_country_mv` as this query will access only `FRESH` portions of the materialized view.

The query is rewritten in terms of `sales_per_country_mv` as follows:

```
SELECT country_name, prod_name, SUM(sum_sales) AS sum_slaes, SUM(cnt) AS cnt
FROM sales_per_country_mv WHERE calendar_year = 2000
  AND country_name IN ('United States of America', 'Japan')
GROUP BY country_name, prod_name;
```

Now consider the following query:

```
SELECT s.country_name,  p.prod_name,
 SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE  s.prod_id = p.prod_id AND s.calendar_year = 1999
  AND s.country_name IN ('Japan', 'India', 'Spain')
GROUP BY s.country_name, p.prod_name;
```

This query accesses partitions `Europe` and `Asia` in `sales_par_list`. Partition `Europe` has been updated, so this query cannot be rewritten in terms of `sales_per_country_mv` as the required data from the materialized view is stale.

You will be able to rewrite after any kinds of updates to `sales_par_list`, that is DMLs, direct loads and Partition Maintenance Operations (PMOPs) if the incoming query accesses `FRESH` parts of the materialized view.

PCT Rewrite and PMARKER

When a partition marker is provided, the query rewrite capabilities are limited to rewrite queries that access whole detail table partitions as all rows from a specific partition have the same pmarker value. That is, if a query accesses a portion of a detail table partition, it is not rewritten even if that data corresponds to a FRESH portion of the materialized view. Now FRESH portions of the materialized view are determined by the pmarker value. To determine which rows of the materialized view are fresh, you associate freshness with the marker value, so all rows in the materialized view with a specific pmarker value are FRESH or are STALE.

The following creates a materialized view has the total amounts sold of every product in each detail table partition of sales_par_list for each year. This materialized view will also depend on detail table products as shown in the following:

```
CREATE MATERIALIZED VIEW sales_per_dt_partition_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT s.calendar_year AS calendar_year, p.prod_name AS prod_name,
       DBMS_MVIEW.PMARKER(s.rowid) pmarker,
       SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE s.prod_id = p.prod_id AND s.calendar_year > 2000
GROUP BY s.calendar_year, DBMS_MVIEW.PMARKER(s.rowid), p.prod_name;
```

The materialized view sales_per_dt_partition_mv provides the sum of sales for each detail table partition. This materialized view supports PCT rewrite against table sales_par_list because the partition marker is in its SELECT and GROUP BY clauses. Table 19–2 lists the partition names and their pmarkers for this example.

Table 19–2 Partition Names and Their Pmarkers

Partition Name	Pmarker
America	1000
Asia	1001
Europe	1002

Then update the table sales_par_list as follows:

```
DELETE FROM sales_par_list WHERE country_name = 'India';
```

You have deleted rows from partition Asia in table sales_par_list. Now sales_per_dt_partition_mv is stale, but PCT rewrite (in ENFORCED and TRUSTED modes) is possible as this materialized view supports PCT (pmarker based) against table sales_par_list.

Now consider the following query:

```
SELECT p.prod_name, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, products p
WHERE  s.prod_id = p.prod_id AND s.calendar_year = 2001 AND
       s.country_name IN ('United States of America', 'Argentina')
GROUP BY p.prod_name;
```

This query can be rewritten in terms of sales_per_dt_partition_mv as all the data corresponding to a detail table partition is accessed, and the materialized view is FRESH with respect to this data. This query accesses all data in partition America, which has not been updated.

The query is rewritten in terms of sales_per_dt_partition_mv as follows:

```
SELECT prod_name, SUM(sum_sales) AS sum_sales, SUM(cnt) AS cnt
FROM sales_per_dt_partition_mv
WHERE calendar_year = 2001 AND pmarker = 1000
GROUP BY prod_name;
```

PCT Rewrite Using Rowid as PMARKER

A materialized view supports PCT rewrite provided a partition key or a partition marker is provided in its SELECT and GROUP BY clause, if there is a GROUP BY clause. You can use the rowids of the partitioned table instead of the pmarker or the partition key. Note that Oracle converts the rowids into pmarkers internally. Consider the following table:

```
CREATE TABLE product_par_list
(prod_id, prod_name, prod_category,
 prod_subcategory, prod_list_price)
 PARTITION BY LIST (prod_category)
 (PARTITION prod_cat1
      VALUES ('Boys', 'Men'),
  PARTITION prod_cat2
      VALUES ('Girls', 'Women'))
 AS
   SELECT prod_id, prod_name, prod_category,
      prod_subcategory, prod_list_price
   FROM products;
```

Let us create the following materialized view on tables, sales_par_list and product_par_list:

```
CREATE MATERIALIZED VIEW  sum_sales_per_category_mv
BUILD IMMEDIATE
REFRESH FORCE ON DEMAND
ENABLE QUERY REWRITE AS
SELECT p.rowid prid, p.prod_category,
       SUM (s.amount_sold) sum_sales, COUNT(*) cnt
FROM sales_par_list s, product_par_list p
WHERE s.prod_id = p.prod_id and s.calendar_year <= 2000
GROUP BY p.rowid, p.prod_category;
```

All the limitations that apply to pmarker rewrite apply here as well. The incoming query should access a whole partition for the query to be rewritten. The following pmarker table is used in this case:

```
product_par_list       pmarker value
----------------       -------------
prod_cat1                     1000
prod_cat2                     1001
prod_cat3                     1002
```

Then update table product_par_list as follows:

```
DELETE FROM product_par_list WHERE prod_name = 'MEN';
```

So sum_sales_per_category_mv is stale with respect to partition prod_list1 from product_par_list.

Now consider the following query:

```
SELECT p.prod_category, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_list s, product_par_list p
```

```
WHERE  s.prod_id = p.prod_id AND p.prod_category IN
       ('Girls', 'Women') AND s.calendar_year <= 2000
GROUP BY p.prod_category;
```

This query can be rewritten in terms of sum_sales_per_category_mv as all the data corresponding to a detail table partition is accessed, and the materialized view is FRESH with respect to this data. This query accesses all data in partition prod_cat2, which has not been updated. Following is the rewritten query in terms of sum_sales_per_category_mv:

```
SELECT prod_category, sum_sales, cnt
FROM sum_sales_per_category_mv WHERE DBMS_MVIEW.PMARKER(srid) IN (1000)
GROUP BY prod_category;
```

Multiple Materialized Views

Query rewrite has been extended to enable the rewrite of a query using multiple materialized views. If query rewrite determines that there is no set of materialized views that returns all of the data, then query rewrite retrieves the remaining data from the base tables.

Query rewrite using multiple materialized views can take advantage of many different types and combinations of rewrite, such as using PCT and IN-lists. The following examples illustrate some of the queries where query rewrite is now possible.

Consider the following two materialized views, cust_avg_credit_mv1 and cust_avg_credit_mv2. cust_avg_credit_mv1 asks for all customers average credit limit for each postal code that were born between the years 1940 and 1950. cust_avg_credit_mv2 asks for customers average credit limit for each postal code that were born after 1950 and before or on 1970.

The materialized views' definitions for this example are as follows:

```
CREATE MATERIALIZED VIEW cust_avg_credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_postal_code, cust_year_of_birth,
       SUM(cust_credit_limit) AS sum_credit,
       COUNT(cust_credit_limit) AS count_credit
FROM customers
WHERE cust_year_of_birth BETWEEN 1940 AND 1950
GROUP BY cust_postal_code, cust_year_of_birth;

CREATE MATERIALIZED VIEW cust_avg_credit_mv2
ENABLE QUERY REWRITE
AS SELECT cust_postal_code, cust_year_of_birth,
       SUM(cust_credit_limit) AS sum_credit,
       COUNT(cust_credit_limit) AS count_credit
FROM customers
WHERE cust_year_of_birth > 1950 AND cust_year_of_birth <= 1970
GROUP BY cust_postal_code, cust_year_of_birth;
```

Query 1: One Matched Interval in Materialized View and Query

Consider a query that asks for all customers average credit limit for each postal code who were born between 1940 and 1970. This query is matched by the interval BETWEEN on cust_year_of_birth.

```
SELECT cust_postal_code, AVG(cust_credit_limit) AS avg_credit
FROM customers c
WHERE cust_year_of_birth BETWEEN 1940 AND 1970
GROUP BY cust_postal_code;
```

The preceding query can be rewritten in terms of these two materialized views to get all the data as follows:

```
SELECT v1.cust_postal_code,
SUM(v1.sum_credit)/SUM(v1.count_credit) AS avg_credit
FROM (SELECT cust_postal_code, sum_credit, count_credit
 FROM cust_avg_credit_mv1
 GROUP BY cust_postal_code
 UNION ALL
 SELECT cust_postal_code, sum_credit, count_credit
 FROM cust_avg_credit_mv2
 GROUP BY cust_postal_code) v1
 GROUP BY v1.cust_postal_code;
```

Note that the UNION ALL query is used in an inline view because of the re-aggregation that needs to take place. Note also how query rewrite was the count aggregate to perform this rollup.

Query 2: Query Outside of Data Contained in Materialized View

When the materialized view goes beyond the range asked by the query, a filter (also called selection) is added to the rewritten query to drop out the unneeded rows returned by the materialized view. This case is illustrated in the following query:

```
SELECT cust_postal_code, SUM(cust_credit_limit) AS sum_credit
FROM customers c
WHERE cust_year_of_birth BETWEEN 1945 AND 1955
GROUP BY cust_postal_code;
```

Query 2 is rewritten as:

```
SELECT v1.cust_postal_code, SUM(v1.sum_credit)
FROM
(SELECT cust_postal_code, SUM(sum_credit) AS sum_credit
FROM cust_avg_credit_mv1
WHERE cust_year_of_birth BETWEEN 1945 AND 1950
GROUP BY cust_postal_code
UNION ALL
SELECT cust_postal_code, SUM(sum_credit) AS sum_credit
FROM cust_birth_mv2
WHERE cust_year_of_birth > 1950 AND cust_year_of_birth <= 1955
GROUP BY cust_postal_code) v1
GROUP BY v1.cust_postal_code;
```

Query 3: Requesting More Data Than is in the Materialized View

What if a query asks for more data than is contained in the two materialized views? It still rewrites using both materialized views and the data in the base table. In the following example, a new set of materialized views without aggregates is defined It will still rewrite using both materialized views and the data in the base table.

```
CREATE MATERIALIZED VIEW cust_birth_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name, cust_year_of_birth
FROM customers WHERE cust_year_of_birth BETWEEN 1940 AND 1950;

CREATE MATERIALIZED VIEW cust_avg_credit_mv2
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name, cust_year_of_birth
FROM customers
WHERE cust_year_of_birth > 1950 AND cust_year_of_birth <= 1970;
```

Our queries now require all customers born between 1940 and 1990.

```
SELECT cust_last_name, cust_first_name
FROM customers c WHERE cust_year_of_birth BETWEEN 1940 AND 1990;
```

Query rewrite needs to access the base table to access the customers that were born after 1970 and before or on 1990. Therefore, Query 3 is rewritten as the following:

```
SELECT cust_last_name, cust_first_name
FROM cust_birth_mv1
UNION ALL
SELECT cust_last_name, cust_first_name
FROM cust_birth_mv2
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers c
WHERE cust_year_of_birth > 1970 AND cust_year_of_birth <= 1990;
```

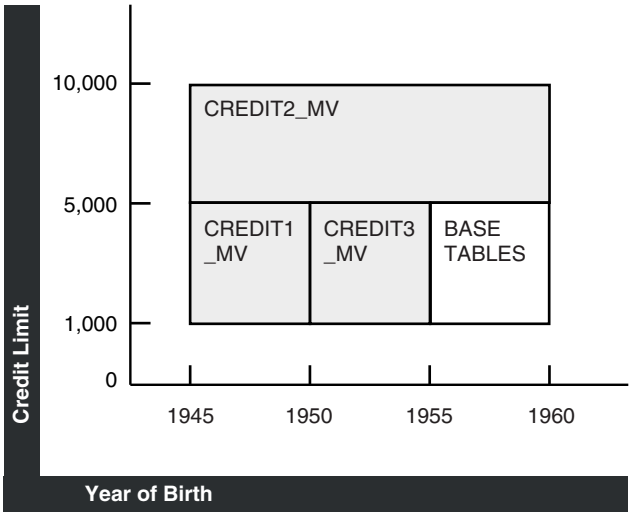
Query 4: Requesting Data on Multiple Selection Columns

Consider the following query, which asks for all customers who have a credit limit between 1,000 and 10,000 and were born between the years 1945 and 1960. This query is a multi-selection query because it is asking for data on multiple selection columns.

```
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_year_of_birth BETWEEN 1945 AND 1960 AND
   cust_credit_limit BETWEEN 1000 AND 10000;
```

Figure 19–5 shows a two-selection query, which can be rewritten with the two-selection materialized views described in the following section.

Figure 19–5 Query Rewrite Using Multiple Materialized Views



The graph in Figure 19–5 illustrates the materialized views that can be used to satisfy this query. `credit_mv1` asks for customers that have credit limits between 1,000 and 5,000 and were born between 1945 and 1950. `credit_mv2` asks for customers that have credit limits > 5,000 and <= 10,000 and were born between 1945 and 1960. `credit_mv3` asks for customers that have credit limits between 1,000 and 5,000 and were born after 1950 and before or on 1955.

The materialized views' definitions for this case are as follows:

```
CREATE MATERIALIZED VIEW credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
    cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
AND cust_year_of_birth BETWEEN 1945 AND 1950;

CREATE MATERIALIZED VIEW credit_mv2
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit > 5000
   AND cust_credit_limit <= 10000 AND cust_year_of_birth
   BETWEEN 1945 AND 1960;

CREATE MATERIALIZED VIEW credit_mv3
ENABLE QUERY REWRITE AS
SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
  AND cust_year_of_birth > 1950 AND cust_year_of_birth <= 1955;
```

Query 4 can be rewritten by using all three materialized views to access most of the data. However, because not all the data can be obtained from these three materialized views, query rewrite also accesses the base tables to retrieve the data for customers who have credit limits between 1,000 and 5,000 and were born between 1955 and 1960. It is rewritten as follows:

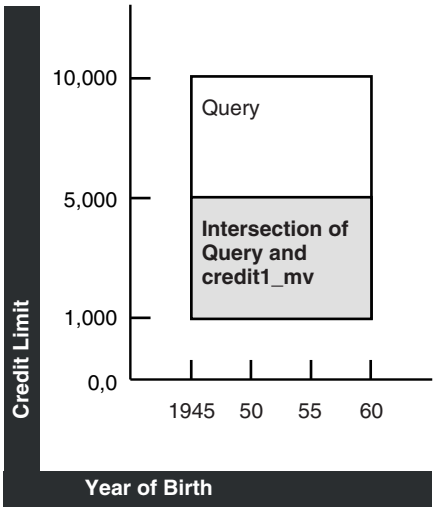
```
SELECT cust_last_name, cust_first_name
FROM credit_mv1
UNION ALL
SELECT cust_last_name, cust_first_name
FROM credit_mv2
UNION ALL
SELECT cust_last_name, cust_first_name
FROM credit_mv3
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
  AND cust_year_of_birth > 1955 AND cust_year_of_birth <= 1960;
```

This example illustrates how a multi-selection query can be rewritten with multiple materialized views. The example was simplified to show no overlapping data among the three materialized views. However, query rewrite can perform similar rewrites.

Query 5: Intervals and Constrained Intervals

This example illustrates how a multi-selection query can be rewritten using a single selection materialized view. In this example, there are two intervals in the query and one constrained interval in the materialized view. It asks for customers that have credit limits between 1,000 and 10,000 and were born between 1945 and 1960. But suppose that `credit_mv1` asks for just customers that have credit limits between 1,000 and 5,000. `credit_mv1` is not constrained by a selection in `cust_year_of_birth`, therefore covering the entire range of birth year values for the query.

Figure 19–6 Constrained Materialized View Selections



The area between the lines in Figure 19–6 represents the data `credit1_mv`.

The new `credit_mv1` is defined as follows:

```
CREATE MATERIALIZED VIEW credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
FROM customers WHERE cust_credit_limit BETWEEN 1000 AND 5000;
```

The query is as follows:

```
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_year_of_birth BETWEEN 1945 AND 1960
  AND cust_credit_limit BETWEEN 1000 AND 10000;
```

And finally the rewritten query is as follows:

```
SELECT cust_last_name, cust_first_name
FROM credit_mv1 WHERE cust_year_of_birth BETWEEN 1945 AND 1960
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_year_of_brith BETWEEN 1945 AND 1960
  AND cust_credit_limit > 5000 AND cust_credit_limit <= 10000;
```

Query 6: Query has Single Column IN-List and Materialized Views have Single Column Intervals

Multiple materialized view query rewrite can process an `IN`-list in the incoming query and rewrite the query in terms of materialized views that have intervals on the same selection column. Given that an `IN`-list represents discrete values in an interval, this rewrite capability is a natural extension to the intervals only scenario described earlier.

The following is an example of a one column `IN`-list selection in the query and one column interval selection in the materialized views. Consider a query that asks for the number of customers for each country who were born in any of the following year: 1945, 1950, 1955, 1960, 1965, 1970 or 1975. This query is constrained by an `IN`-list on `cust_year_of_birth`.

```
SELECT c2.country_name, count(c1.country_id)
FROM customers c1, countries c2
```

```
WHERE c1.country_id = c2.country_id AND
   c1.cust_year_of_birth IN (1945, 1950, 1955, 1960, 1965, 1970, 1975)
GROUP BY c2.country_name;
```

Consider the following two materialized views. `cust_country_birth_mv1` asks for the number of customers for each country that were born between the years 1940 and 1950. `cust_country_birth_mv2` asks for the number of customers for each country that were born after 1950 and before or on 1970. The preceding query can be rewritten in terms of these two materialized views to get the total number of customers for each country born in 1945, 1950, 1955, 1960, 1965 and 1970. The base table access is required to obtain the number of customers that were born in 1975.

The materialized views' definitions for this example are as follows:

```
CREATE MATERIALIZED VIEW cust_country_birth_mv1
ENABLE QUERY REWRITE
AS SELECT c2.country_name, c1.cust_year_of_birth,
 COUNT(c1.country_id) AS count_customers
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND
      cust_year_of_birth BETWEEN 1940 AND 1950
GROUP BY c2.country_name, c1.cust_year_of_birth;

CREATE MATERIALIZED VIEW cust_country_birth_mv2
ENABLE QUERY REWRITE
AS SELECT c2.country_name, c1.cust_year_of_birth,
 COUNT(c1.country_id) AS count_customers
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND cust_year_of_birth > 1950
AND cust_year_of_birth <= 1970
GROUP BY c2.country_name, c1.cust_year_of_birth;
```

So, Query 6 is rewritten as:

```
SELECT v1.country_name, SUM(v1.count_customers)
FROM (SELECT country_name, SUM(count_customers) AS count_customers
FROM cust_country_birth_mv1
WHERE cust_year_of_birth IN (1945, 1950)
GROUP BY country_name
UNION ALL
SELECT country_name, SUM(count_customers) AS count_customers
FROM cust_country_birth_mv2
WHERE cust_year_of_birth IN (1955, 1960, 1965, 1970)
GROUP BY country_name
UNION ALL
SELECT c2.country_name, COUNT(c1.country_id) AS count_customers
FROM customers c1, countries c2
WHERE c1.country_id = c2.country_id AND cust_year_of_birth IN (1975)
GROUP BY c2.country_name) v1
GROUP BY v1.country_name;
```

Query 7: PCT Rewrite with Multiple Materialized Views

Rewrite with multiple materialized views can also take advantage of PCT rewrite. PCT rewrite refers to the capability of rewriting a query with only the fresh portions of a materialized view when the materialized view is stale. This feature is used in `ENFORCED` or `TRUSTED` integrity modes, and with multiple materialized view rewrite, it can use the fresh portions of the materialized view to get the fresh data from it, and go to the base table to get the stale data. So the rewritten query will `UNION ALL` only the fresh data from one or more materialized views and obtain the rest of the data

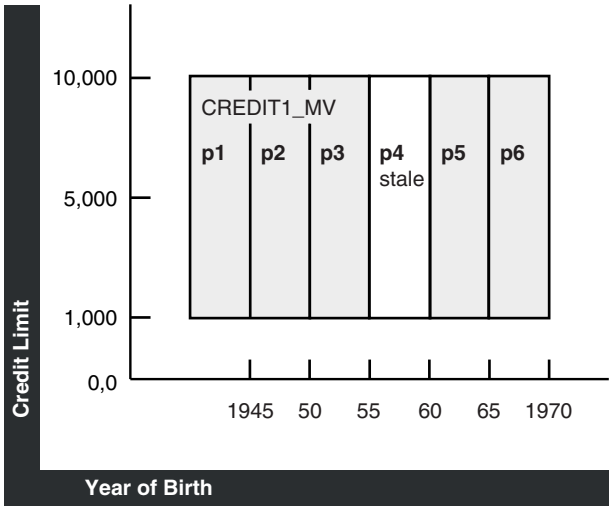
from the base tables to answer the query. Therefore, all the PCT rules and conditions apply here as well. The materialized view should be PCT enabled and the changes made to the base table should be such that the fresh and stale portions of the materialized view can be clearly identified.

This example assumes you have a query that asks for customers who have credit limits between 1,000 and 10,000 and were born between 1945 and 1964. Also, the customer table is partitioned by `cust_date_of_birth` and there is a PCT-enabled materialized view called `credit_mv1` that also asks for customers who have a credit limit between 1,000 and 10,000 and were born between 1945 and 1964.

```
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_credit_limit BETWEEN 1000 AND 10000;
```

In Figure 19–7, the diagram illustrates those regions of the materialized view that are fresh (dark) and stale (light) with respect to the base table partitions `p1`-`p6`.

Figure 19–7 PCT and Multiple Materialized View Rewrite



Let us say that we are in `ENFORCED` mode and that `p1`, `p2`, `p3`, `p5`, and `p6` of the customer table are fresh and partition `p4` is stale. This means that all partitions of `credit_mv1` cannot be used to answer the query. The rewritten query must get the results for customer partition `p4` from some other materialized view or as shown in this example, from the base table. Below, we can see part of the table definition for the `customers` table showing how the table is partitioned:

```
CREATE TABLE customers
(PARTITION BY RANGE (cust_year_of_birth)
 PARTITION p1 VALUES LESS THAN (1945),
 PARTITION p2 VALUES LESS THAN (1950),
 PARTITION p3 VALUES LESS THAN (1955),
 PARTITION p4 VALUES LESS THAN (1960),
 PARTITION p5 VALUES LESS THAN (1965),
 PARTITION p6 VALUES LESS THAN (1970);
```

The materialized view definition for the preceding example is as follows:

```
CREATE MATERIALIZED VIEW credit_mv1
ENABLE QUERY REWRITE
AS SELECT cust_last_name, cust_first_name,
   cust_credit_limit, cust_year_of_birth
```

```
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 10000
AND cust_year_of_birth BETWEEN 1945 AND 1964;
```

Note that this materialized view is PCT enabled with respect to table `customers`.

The rewritten query is as follows:

```
SELECT cust_last_name, cust_first_name FROM credit_mv1
WHERE cust_credit_limit BETWEEN 1000 AND 10000 AND
   (cust_year_of_birth >= 1945 AND cust_year_of_birth < 1955 OR
    cust_year_of_birth BETWEEN 1945 AND 1964)
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers WHERE cust_credit_limit BETWEEN 1000 AND 10000
  AND cust_year_of_birth < 1960 AND cust_year_of_birth >= 1955;
```

Other Query Rewrite Considerations

The following discusses some of the other cases when query rewrite is possible:

- Query Rewrite Using Nested Materialized Views
- Query Rewrite in the Presence of Inline Views
- Query Rewrite Using Remote Tables
- Query Rewrite in the Presence of Duplicate Tables
- Query Rewrite Using Date Folding
- Query Rewrite Using View Constraints
- Query Rewrite Using Set Operator Materialized Views
- Query Rewrite in the Presence of Grouping Sets
- Query Rewrite in the Presence of Window Functions
- Query Rewrite and Expression Matching
- Cursor Sharing and Bind Variables
- Handling Expressions in Query Rewrite

Query Rewrite Using Nested Materialized Views

Query rewrite attempts to iteratively take advantage of nested materialized views. Oracle Database first tries to rewrite a query with materialized views having aggregates and joins, then with a materialized view containing only joins. If any of the rewrites succeeds, Oracle repeats that process again until no rewrites are found. For example, assume that you had created materialized views `join_sales_time_product_mv` and `sum_sales_time_product_mv` as in the following:

```
CREATE MATERIALIZED VIEW join_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, p.prod_name, t.time_id, t.week_ending_day,
       s.channel_id, s.promo_id, s.cust_id, s.amount_sold
FROM   sales s, products p, times t
WHERE  s.time_id=t.time_id AND s.prod_id = p.prod_id;

CREATE MATERIALIZED VIEW sum_sales_time_product_mv
ENABLE QUERY REWRITE AS
SELECT mv.prod_name, mv.week_ending_day, COUNT(*) cnt_all,
```

```
       SUM(mv.amount_sold) sum_amount_sold,
       COUNT(mv.amount_sold) cnt_amount_sold
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

Then consider the following query:

```
SELECT p.prod_name, t.week_ending_day, SUM(s.amount_sold)
FROM sales s, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_name, t.week_ending_day;
```

Oracle finds that `join_sales_time_product_mv` is eligible for rewrite. The rewritten query has this form:

```
SELECT mv.prod_name, mv.week_ending_day, SUM(mv.amount_sold)
FROM join_sales_time_product_mv mv
GROUP BY mv.prod_name, mv.week_ending_day;
```

Because a rewrite occurred, Oracle tries the process again. This time, the query can be rewritten with single-table aggregate materialized view `sum_sales_store_time` into the following form:

```
SELECT mv.prod_name, mv.week_ending_day, mv.sum_amount_sold
FROM sum_sales_time_product_mv mv;
```

Query Rewrite in the Presence of Inline Views

Oracle Database supports query rewrite with inline views in two ways:

- when the text from the inline views in the materialized view exactly matches the text in the request query
- when the request query contains inline views that are equivalent to the inline views in the materialized view

Two inline views are considered equivalent if their `SELECT` lists and `GROUP BY` lists are equivalent, `FROM` clauses contain the same or equivalent objects, their join graphs, including all the selections in the `WHERE` clauses are equivalent and their `HAVING` clauses are equivalent.

The following examples illustrate how a query with an inline view can rewrite with a materialized view using text match and general inline view rewrites. Consider the following materialized view that contains an inline view:

```
CREATE MATERIALIZED VIEW SUM_SALES_MV
ENABLE QUERY REWRITE AS
SELECT mv_iv.prod_id, mv_iv.cust_id,
sum(mv_iv.amount_sold) sum_amount_sold
FROM (SELECT prod_id, cust_id, amount_sold
FROM sales, products
WHERE sales.prod_id = products.prod_id) MV_IV
GROUP BY mv_iv.prod_id, mv_iv.cust_id;
```

The following query has an inline view whose text matches exactly with that of the materialized view's inline view. Hence, the query inline view is internally replaced with the materialized view's inline view so that the query can be rewritten:

```
SELECT iv.prod_id, iv.cust_id,
SUM(iv.amount_sold) sum_amount_sold
FROM (SELECT prod_id, cust_id, amount_sold
FROM sales, products
```

```
WHERE sales.prod_id = products.prod_id) IV
GROUP BY iv.prod_id, iv.cust_id;
```

The following query has an inline view that does not have exact text match with the inline view in the preceding materialized view. Note that the join predicate in the query inline view is switched. Even though this query does not textually match with that of the materialized view's inline view, query rewrite identifies the query's inline view as equivalent to the materialized view's inline view. As before, the query inline view will be internally replaced with the materialized view's inline view so that the query can be rewritten.

```
SELECT iv.prod_id, iv.cust_id,
SUM(iv.amount_sold) sum_amount_sold
FROM (SELECT prod_id, cust_id, amount_sold
FROM sales, products
WHERE products.prod_id = sales.prod_id) IV
GROUP BY iv.prod_id, iv.cust_id;
```

Both of these queries are rewritten with `SUM_SALES_MV` as follows:

```
SELECT prod_id, cust_id, sum_amount_sold
FROM SUM_SALES_MV;
```

General inline view rewrite is not supported for queries that contain set operators, `GROUPING SET` clauses, nested subqueries, nested inline views, and remote tables.

Query Rewrite Using Remote Tables

Oracle Database supports query rewrite with materialized views that reference tables at a single remote database site. Note that the materialized view should be present at the site where the query is being issued. Because any remote table update cannot be propagated to the local site simultaneously, query rewrite only works in the `stale_tolerated` mode. Whenever a query contains columns that are not found in the materialized view, it uses a technique called join back to rewrite the query. However, if the join back table is not found at the local site, query rewrite does not take place. Also, because the constraint information of the remote tables is not available at the remote site, query rewrite does not make use of any constraint information.

The following query contains tables that are found at a single remote site:

```
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales@remotedbl s, products@remotedbl p, times@remotedbl t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;
```

The following materialized view is present at the local site, but it references tables that are all found at the remote site:

```
CREATE MATERIALIZED VIEW sum_sales_prod_week_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_id, t.week_ending_day, s.cust_id,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales@remotedbl s, products@remotedbl p, times@remotedbl t
WHERE s.time_id=t.time_id AND s.prod_id=p.prod_id
GROUP BY p.prod_id, t.week_ending_day, s.cust_id;
```

Even though the query references remote tables, it is rewritten using the previous materialized view as follows:

```
SELECT prod_id, week_ending_day, cust_id,  sum_amount_sold
```

```
FROM sum_sales_prod_week_mv;
```

Query Rewrite in the Presence of Duplicate Tables

Oracle Database accomplishes query rewrite of queries that contain multiple references to the same tables, or self joins by employing two different strategies. Using the first strategy, you need to ensure that the query and the materialized view definitions have the same aliases for the multiple references to a table. If you do not provide a matching alias, Oracle tries the second strategy, where the joins in the query and the materialized view are compared to match the multiple references in the query to the multiple references in the materialized view.

The following is an example of a materialized view and a query. In this example, the query is missing a reference to a column in a table so an exact text match does not work. General query rewrite can occur, however, because the aliases for the table references match.

To demonstrate the self-join rewriting possibility with the `sh` sample schema, the following addition is assumed to include the actual shipping and payment date in the fact table, referencing the same dimension table times. This is for demonstration purposes only and does not return any results:

```
ALTER TABLE sales ADD (time_id_ship DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_book_fk FOREIGN key (time_id_ship)
 REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_book_fk RELY;
ALTER TABLE sales ADD (time_id_paid DATE);
ALTER TABLE sales ADD (CONSTRAINT time_id_paid_fk FOREIGN KEY (time_id_paid)
 REFERENCES times(time_id) ENABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_id_paid_fk RELY;
```

Now, you can define a materialized view as follows:

```
CREATE MATERIALIZED VIEW sales_shipping_lag_mv
ENABLE QUERY REWRITE AS
SELECT t1.fiscal_week_number, s.prod_id,
         t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_ship;
```

The following query fails the exact text match test but is rewritten because the aliases for the table references match:

```
SELECT s.prod_id, t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_ship;
```

Note that Oracle Database performs other checks to ensure the correct match of an instance of a multiply instanced table in the request query with the corresponding table instance in the materialized view. For instance, in the following example, Oracle correctly determines that the matching alias names used for the multiple instances of table `times` does not establish a match between the multiple instances of table `times` in the materialized view.

The following query cannot be rewritten using `sales_shipping_lag_mv`, even though the alias names of the multiply instanced table `time` match because the joins are not compatible between the instances of `time` aliased by `t2`:

```
SELECT s.prod_id, t2.fiscal_week_number - t1.fiscal_week_number AS lag
FROM times t1, sales s, times t2
WHERE t1.time_id = s.time_id AND t2.time_id = s.time_id_paid;
```

This request query joins the instance of the time table aliased by t2 on the s.time_id_paid column, while the materialized views joins the instance of the times table aliased by t2 on the s.time_id_ship column. Because the join conditions differ, Oracle correctly determines that rewrite cannot occur.

The following query does not have any matching alias in the materialized view, sales_shipping_lag_mv, for the table, times. But query rewrite now compares the joins between the query and the materialized view and correctly match the multiple instances of times.

```
SELECT s.prod_id, x2.fiscal_week_number - x1.fiscal_week_number AS lag
FROM times x1, sales s, times x2
WHERE x1.time_id = s.time_id AND x2.time_id = s.time_id_ship;
```

Query Rewrite Using Date Folding

Date folding rewrite is a specific form of expression matching rewrite. In this type of rewrite, a date range in a query is folded into an equivalent date range representing higher date granules. The resulting expressions representing higher date granules in the folded date range are matched with equivalent expressions in a materialized view. The folding of date range into higher date granules such as months, quarters, or years is done when the underlying data type of the column is an Oracle DATE. The expression matching is done based on the use of canonical forms for the expressions.

DATE is a built-in data type which represents ordered time units such as seconds, days, and months, and incorporates a time hierarchy (second -> minute -> hour -> day -> month -> quarter -> year). This hard-coded knowledge about DATE is used in folding date ranges from lower-date granules to higher-date granules. Specifically, folding a date value to the beginning of a month, quarter, year, or to the end of a month, quarter, year is supported. For example, the date value 1-jan-1999 can be folded into the beginning of either year 1999 or quarter 1999-1 or month 1999-01. And, the date value 30-sep-1999 can be folded into the end of either quarter 1999-03 or month 1999-09.

Note: Due to the way date folding works, you should be careful when using BETWEEN and date columns. The best way to use BETWEEN and date columns is to increment the later date by 1. In other words, instead of using date_col BETWEEN '1-jan-1999' AND '30-jun-1999', you should use date_col BETWEEN '1-jan-1999' AND '1-jul-1999'. You could also use the TRUNC function to get the equivalent result, as in TRUNC(date_col) BETWEEN '1-jan-1999' AND '30-jun-1999'. TRUNC will, however, strip time values.

Because date values are ordered, any range predicate specified on date columns can be folded from lower level granules into higher level granules provided the date range represents an integral number of higher level granules. For example, the range predicate date_col >= '1-jan-1999' AND date_col < '30-jun-1999' can be folded into either a month range or a quarter range using the TO_CHAR function, which extracts specific date components from a date value.

The advantage of aggregating data by folded date values is the compression of data achieved. Without date folding, the data is aggregated at the lowest granularity level, resulting in increased disk space for storage and increased I/O to scan the materialized view.

Consider a query that asks for the sum of sales by product types for the year 1998:

```
SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id AND s.time_id >= TO_DATE('01-jan-1998', 'dd-mon-yyyy')
  AND s.time_id <   TO_DATE('01-jan-1999', 'dd-mon-yyyy')
GROUP BY p.prod_category;

CREATE MATERIALIZED VIEW sum_sales_pcat_monthly_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, TO_CHAR(s.time_id,'YYYY-MM') AS month,
       SUM(s.amount_sold) AS sum_amount
FROM sales s, products p
WHERE s.prod_id=p.prod_id
GROUP BY p.prod_category, TO_CHAR(s.time_id, 'YYYY-MM');

SELECT p.prod_category, SUM(s.amount_sold)
FROM sales s, products p
WHERE s.prod_id=p.prod_id
AND TO_CHAR(s.time_id, 'YYYY-MM') >= '01-jan-1998'
AND TO_CHAR(s.time_id, 'YYYY-MM') < '01-jan-1999'
GROUP BY p.prod_category;

SELECT mv.prod_category, mv.sum_amount
FROM sum_sales_pcat_monthly_mv mv
WHERE month >= '01-jan-1998' AND month < '01-jan-1999';
```

The range specified in the query represents an integral number of years, quarters, or months. Assume that there is a materialized view `mv3` that contains pre-summarized sales by `prod_type` and is defined as follows:

```
CREATE MATERIALIZED VIEW mv3
ENABLE QUERY REWRITE AS
SELECT prod_name, TO_CHAR(sales.time_id,'yyyy-mm')
   AS month, SUM(amount_sold) AS sum_sales
FROM sales, products WHERE sales.prod_id = products.prod_id
GROUP BY prod_name, TO_CHAR(sales_time_id, 'yyyy-mm');
```

The query can be rewritten by first folding the date range into the month range and then matching the expressions representing the months with the month expression in `mv3`. This rewrite is shown in two steps (first folding the date range followed by the actual rewrite).

```
SELECT prod_name, SUM(amount_sold) AS sum_sales
FROM sales, products
WHERE sales.prod_id = products.prod_id AND TO_CHAR(sales.time_id, 'yyyy-mm') >=
      TO_CHAR('01-jan-1998', 'yyyy-mm') AND TO_CHAR(sales.time_id, '01-jan-1999',
 'yyyy-mm') < TO_CHAR(TO_DATE(''01-jan-1999'', ''dd-mon-yyyy''), ''yyyy-mm'')
GROUP BY prod_name;

SELECT prod_name, sum_sales
FROM mv3 WHERE month >=
      TO_CHAR(TO_DATE('01-jan-1998', 'dd-mon-yyyy'), 'yyyy-mm')
  AND month < TO_CHAR(TO_DATE('01-jan-1999', 'dd-mon-yyyy'), 'yyyy-mm');
```

If `mv3` had pre-summarized sales by `prod_name` and year instead of `prod_name` and month, the query could still be rewritten by folding the date range into year range and then matching the year expressions.

Query Rewrite Using View Constraints

Data warehouse applications recognize multi-dimensional cubes in the database by identifying integrity constraints in the relational schema. Integrity constraints represent primary and foreign key relationships between fact and dimension tables. By querying the data dictionary, applications can recognize integrity constraints and hence the cubes in the database. However, this does not work in an environment where database administrators, for schema complexity or security reasons, define views on fact and dimension tables. In such environments, applications cannot identify the cubes properly. By allowing constraint definitions between views, you can propagate base table constraints to the views, thereby allowing applications to recognize cubes even in a restricted environment.

View constraint definitions are declarative in nature, but operations on views are subject to the integrity constraints defined on the underlying base tables, and constraints on views can be enforced through constraints on base tables. Defining constraints on base tables is necessary, not only for data correctness and cleanliness, but also for materialized view query rewrite purposes using the original base objects.

Materialized view rewrite extensively uses constraints for query rewrite. They are used for determining lossless joins, which, in turn, determine if joins in the materialized view are compatible with joins in the query and thus if rewrite is possible.

`DISABLE NOVALIDATE` is the only valid state for a view constraint. However, you can choose `RELY` or `NORELY` as the view constraint state to enable more sophisticated query rewrites. For example, a view constraint in the `RELY` state allows query rewrite to occur when the query integrity level is set to `TRUSTED`. Table 19–3 illustrates when view constraints are used for determining lossless joins.

Note that view constraints cannot be used for query rewrite integrity level `ENFORCED`. This level enforces the highest degree of constraint enforcement `ENABLE VALIDATE`.

Table 19–3 View Constraints and Rewrite Integrity Modes

Constraint States	RELY	NORELY
`ENFORCED`	No	No
`TRUSTED`	Yes	No
`STALE_TOLERATED`	Yes	No

Example 19–10 View Constraints

To demonstrate the rewrite capabilities on views, you need to extend the `sh` sample schema as follows:

```
CREATE VIEW time_view AS
SELECT time_id, TO_NUMBER(TO_CHAR(time_id, 'ddd')) AS day_in_year FROM times;
```

You can now establish a foreign key/primary key relationship (in `RELY` mode) between the view and the fact table, and thus rewrite takes place as described in Table 19–3, by adding the following constraints. Rewrite will then work for example in `TRUSTED` mode.

```
ALTER VIEW time_view ADD (CONSTRAINT time_view_pk
   PRIMARY KEY (time_id) DISABLE NOVALIDATE);
ALTER VIEW time_view MODIFY CONSTRAINT time_view_pk RELY;
ALTER TABLE sales ADD (CONSTRAINT time_view_fk FOREIGN KEY (time_id)
   REFERENCES time_view(time_id) DISABLE NOVALIDATE);
ALTER TABLE sales MODIFY CONSTRAINT time_view_fk RELY;
```

Consider the following materialized view definition:

```
CREATE MATERIALIZED VIEW sales_pcat_cal_day_mv
ENABLE QUERY REWRITE AS
SELECT p.prod_category, t.day_in_year, SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s, products p
WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
GROUP BY p.prod_category, t.day_in_year;
```

The following query, omitting the dimension table `products`, is also rewritten without the primary key/foreign key relationships, because the suppressed join between `sales` and `products` is known to be lossless.

```
SELECT t.day_in_year, SUM(s.amount_sold) AS sum_amount_sold
FROM time_view t, sales s WHERE t.time_id = s.time_id
GROUP BY t.day_in_year;
```

However, if the materialized view `sales_pcat_cal_day_mv` were defined only in terms of the view `time_view`, then you could not rewrite the following query, suppressing then join between `sales` and `time_view`, because there is no basis for losslessness of the delta materialized view join. With the additional constraints as shown previously, this query will also rewrite.

```
SELECT p.prod_category, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p WHERE p.prod_id = s.prod_id
GROUP BY p.prod_category;
```

To undo the changes you have made to the `sh` schema, issue the following statements:

```
ALTER TABLE sales DROP CONSTRAINT time_view_fk;
DROP VIEW time_view;
```

View Constraints Restrictions

If the referential constraint definition involves a view, that is, either the foreign key or the referenced key resides in a view, the constraint can only be in `DISABLE NOVALIDATE` mode.

A `RELY` constraint on a view is allowed only if the referenced `UNIQUE` or `PRIMARY KEY` constraint in `DISABLE NOVALIDATE` mode is also a `RELY` constraint.

The specification of `ON DELETE` actions associated with a referential Integrity constraint, is not allowed (for example, `DELETE` cascade). However, `DELETE`, `UPDATE`, and `INSERT` operations are allowed on views and their base tables as view constraints are in `DISABLE NOVALIDATE` mode.

Query Rewrite Using Set Operator Materialized Views

You can use query rewrite with materialized views that contain set operators. In this case, the query and materialized view do not have to match textually for rewrite to occur. As an example, consider the following materialized view, which uses the postal codes for male customers from San Francisco or Los Angeles:

```
CREATE MATERIALIZED VIEW cust_male_postal_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' AND c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
```

```
FROM customers c
WHERE c.cust_gender = 'M' AND c.cust_city = 'Los Angeles';
```

If you have the following query, which displays the postal codes for male customers from San Francisco or Los Angeles:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco' AND c.cust_gender = 'M';
```

The rewritten query will be the following:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_male_postal_mv mv;
```

The rewritten query has dropped the UNION ALL and replaced it with the materialized view. Normally, query rewrite has to use the existing set of general eligibility rules to determine if the SELECT subselections under the UNION ALL are equivalent in the query and the materialized view.

If, for example, you have a query that retrieves the postal codes for male customers from San Francisco, Palmdale, or Los Angeles, the same rewrite can occur as in the previous example but query rewrite must keep the UNION ALL with the base tables, as in the following:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco' AND c.cust_gender = 'M';
```

The rewritten query will be:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_male_postal_mv mv
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Palmdale' AND c.cust_gender = 'M';
```

So query rewrite detects the case where a subset of the UNION ALL can be rewritten using the materialized view cust_male_postal_mv.

UNION, UNION ALL, and INTERSECT are commutative, so query rewrite can rewrite regardless of the order the subselects are found in the query or materialized view. However, MINUS is not commutative. A MINUS B is not equivalent to B MINUS A. Therefore, the order in which the subselects appear under the MINUS operator in the query and the materialized view must be in the same order for rewrite to happen. As an example, consider the case where there exists an old version of the customer table called customer_old and you want to find the difference between the old one and the current customer table only for male customers who live in London. That is, you

want to find those customers in the current one that were not in the old one. The following example shows how this is done using a MINUS operator:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Los Angeles' AND c.cust_gender = 'M'
MINUS
SELECT c.cust_city, c.cust_postal_code
FROM customers_old c
WHERE c.cust_city = 'Los Angeles' AND c.cust_gender = 'M';
```

Switching the subselects would yield a different answer. This illustrates that MINUS is not commutative.

UNION ALL Marker

If a materialized view contains one or more UNION ALL operators, it can also include a UNION ALL marker. The UNION ALL marker is used to identify from which UNION ALL subselect each row in the materialized view originates. Query rewrite can use the marker to distinguish what rows coming from the materialized view belong to a certain UNION ALL subselect. This is useful if the query needs only a subset of the data from the materialized view or if the subselects of the query do not textually match with the subselects of the materialized view. As an example, the following query retrieves the postal codes for male customers from San Francisco and female customers from Los Angeles:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' and c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'F' and c.cust_city = 'Los Angeles';
```

The query can be answered using the following materialized view:

```
CREATE MATERIALIZED VIEW cust_postal_mv
ENABLE QUERY REWRITE AS
SELECT 1 AS marker, c.cust_gender, c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'Los Angeles'
UNION ALL
SELECT 2 AS marker, c.cust_gender, c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city = 'San Francisco';
```

The rewritten query is as follows:

```
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_postal_mv mv
WHERE mv.marker = 2 AND mv.cust_gender = 'M'
UNION ALL
SELECT mv.cust_city, mv.cust_postal_code
FROM cust_postal_mv mv
WHERE mv.marker = 1 AND mv.cust_gender = 'F';
```

The WHERE clause of the first subselect includes `mv.marker = 2` and `mv.cust_gender = 'M'`, which selects only the rows that represent male customers in the second subselect of the UNION ALL. The WHERE clause of the second subselect includes `mv.marker = 1` and `mv.cust_gender = 'F'`, which selects only those rows that

represent female customers in the first subselect of the UNION ALL. Note that query rewrite cannot take advantage of set operators that drop duplicate or distinct rows. For example, UNION drops duplicates so query rewrite cannot tell what rows have been dropped, as in the following:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'M' and c.cust_city = 'San Francisco'
UNION ALL
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_gender = 'F' and c.cust_city = 'Los Angeles';
```

The rewritten query using UNION ALL markers is as follows:

```
SELECT c.cust_city, c.cust_postal_code
FROM customers c
WHERE c.cust_city= 'Palmdale' AND c.cust_gender ='M'
UNION ALL
SELECT mv.cust_city, mv.cust_postal_code

FROM cust_postal_mv mv
WHERE mv.marker = 2 AND mv.cust_gender = 'M'
UNION ALL
  SELECT mv.cust_city, mv.cust_postal_code
  FROM cust_postal_mv mv
  WHERE mv.marker = 1 AND mv.cust_gender = 'F';
```

The rules for using a marker are that it must:

- Be a constant number or string and be the same data type for all UNION ALL subselects.
- Yield a constant, distinct value for each UNION ALL subselect. You cannot reuse the same value in multiple subselects.
- Be in the same ordinal position for all subselects.

Query Rewrite in the Presence of Grouping Sets

This section discusses various considerations for using query rewrite with grouping sets.

Query Rewrite When Using GROUP BY Extensions

Several extensions to the GROUP BY clause in the form of GROUPING SETS, CUBE, ROLLUP, and their concatenation are available. These extensions enable you to selectively specify the groupings of interest in the GROUP BY clause of the query. For example, the following is a typical query with grouping sets:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS ((p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory));
```

The term **base grouping** for queries with GROUP BY extensions denotes all unique expressions present in the GROUP BY clause. In the previous query, the following grouping (p.prod_subcategory, t.calendar_month_desc, c.cust_city) is a base grouping.

The extensions can be present in user queries and in the queries defining materialized views. In both cases, materialized view rewrite applies and you can distinguish rewrite capabilities into the following scenarios:

- Materialized View has Simple GROUP BY and Query has Extended GROUP BY
- Materialized View has Extended GROUP BY and Query has Simple GROUP BY
- Both Materialized View and Query Have Extended GROUP BY

Materialized View has Simple GROUP BY and Query has Extended GROUP BY When a query contains an extended GROUP BY clause, it can be rewritten with a materialized view if its base grouping can be rewritten using the materialized view as listed in the rewrite rules explained in "When Does Oracle Rewrite a Query?" on page 18-2. For example, in the following query:

```
SELECT p.prod_subcategory, t.calendar_month_desc, c.cust_city,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c, products p, times t
WHERE s.time_id=t.time_id AND s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_subcategory, t.calendar_month_desc),
   (c.cust_city, p.prod_subcategory));
```

The base grouping is (p.prod_subcategory, t.calendar_month_desc, c.cust_city, p.prod_subcategory)) and, consequently, Oracle can rewrite the query using sum_sales_pscat_month_city_mv as follows:

```
SELECT mv.prod_subcategory, mv.calendar_month_desc, mv.cust_city,
  SUM(mv.sum_amount_sold) AS sum_amount_sold
FROM sum_sales_pscat_month_city_mv mv
GROUP BY GROUPING SETS
((mv.prod_subcategory, mv.calendar_month_desc),
  (mv.cust_city, mv.prod_subcategory));
```

A special situation arises if the query uses the EXPAND_GSET_TO_UNION hint. See "Hint for Queries with Extended GROUP BY" on page 19-51 for an example of using EXPAND_GSET_TO_UNION.

Materialized View has Extended GROUP BY and Query has Simple GROUP BY In order for a materialized view with an extended GROUP BY to be used for rewrite, it must satisfy two additional conditions:

- It must contain a grouping distinguisher, which is the GROUPING_ID function on all GROUP BY expressions. For example, if the GROUP BY clause of the materialized view is GROUP BY CUBE(a, b), then the SELECT list should contain GROUPING_ID(a, b).
- The GROUP BY clause of the materialized view should not result in any duplicate groupings. For example, GROUP BY GROUPING SETS ((a, b), (a, b)) would disqualify a materialized view from general rewrite.

A materialized view with an extended GROUP BY contains multiple groupings. Oracle finds the grouping with the lowest cost from which the query can be computed and uses that for rewrite. For example, consider the following materialized view:

```
CREATE MATERIALIZED VIEW sum_grouping_set_mv
```

```
ENABLE QUERY REWRITE AS
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city,
  GROUPING_ID(p.prod_category,p.prod_subcategory,
              c.cust_state_province,c.cust_city) AS gid,
  SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory, c.cust_state_province, c.cust_city),
  (p.prod_category, p.prod_subcategory));
```

In this case, the following query is rewritten:

```
SELECT p.prod_subcategory, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, c.cust_city;
```

This query is rewritten with the closest matching grouping from the materialized view. That is, the `(prod_category, prod_subcategory, cust_city)` grouping:

```
SELECT prod_subcategory, cust_city, SUM(sum_amount_sold) AS sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
GROUP BY prod_subcategory, cust_city;
```

Both Materialized View and Query Have Extended GROUP BY When both materialized view and the query contain `GROUP BY` extensions, Oracle uses two strategies for rewrite: grouping match and `UNION ALL` rewrite. First, Oracle tries grouping match. The groupings in the query are matched against groupings in the materialized view and if all are matched with no rollup, Oracle selects them from the materialized view. For example, consider the following query:

```
SELECT p.prod_category, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
((p.prod_category, p.prod_subcategory, c.cust_city),
  (p.prod_category, p.prod_subcategory));
```

This query matches two groupings from `sum_grouping_set_mv` and Oracle rewrites the query as the following:

```
SELECT prod_subcategory, cust_city, sum_amount_sold
FROM sum_grouping_set_mv
WHERE gid = grouping identifier of (prod_category,prod_subcategory, cust_city)
   OR gid = grouping identifier of (prod_category,prod_subcategory)
```

If grouping match fails, Oracle tries a general rewrite mechanism called `UNION ALL` rewrite. Oracle first represents the query with the extended `GROUP BY` clause as an equivalent `UNION ALL` query. Every grouping of the original query is placed in a separate `UNION ALL` branch. The branch will have a simple `GROUP BY` clause. For example, consider this query:

```
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province,
  t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
```

```
GROUP BY GROUPING SETS
((p.prod_subcategory, t.calendar_month_desc),
  (t.calendar_month_desc),
  (p.prod_category, p.prod_subcategory, c.cust_state_province),
  (p.prod_category, p.prod_subcategory));
```

This is first represented as `UNION ALL` with four branches:

```
SELECT null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_subcategory, t.calendar_month_desc
UNION ALL
  SELECT null, null, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY t.calendar_month_desc
UNION ALL
SELECT p.prod_category, p.prod_subcategory, c.cust_state_province,
    null, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT p.prod_category, p.prod_subcategory, null,
    null, SUM(s.amount_sold) AS sum_amount_sold
  FROM sales s, products p, customers c, times t
  WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
  GROUP BY p.prod_category, p.prod_subcategory;
```

Each branch is then rewritten separately using the rules from "When Does Oracle Rewrite a Query?" on page 18-2. Using the materialized view `sum_grouping_set_mv`, Oracle can rewrite only branches three (which requires materialized view rollup) and four (which matches the materialized view exactly). The unrewritten branches will be converted back to the extended `GROUP BY` form. Thus, eventually, the query is rewritten as:

```
SELECT null, p.prod_subcategory, null,
    t.calendar_month_desc, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.prod_id = p.prod_id AND s.cust_id = c.cust_id
GROUP BY GROUPING SETS
  ((p.prod_subcategory, t.calendar_month_desc),
    (t.calendar_month_desc),)
UNION ALL
  SELECT prod_category, prod_subcategory, cust_state_province,
    null, SUM(sum_amount_sold) AS sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory, cust_city)>
  GROUP BY p.prod_category, p.prod_subcategory, c.cust_state_province
UNION ALL
  SELECT prod_category, prod_subcategory, null,
    null, sum_amount_sold
  FROM sum_grouping_set_mv
  WHERE gid = <grouping id of (prod_category,prod_subcategory)>
```

Note that a query with extended GROUP BY is represented as an equivalent UNION ALL and recursively submitted for rewrite optimization. The groupings that cannot be rewritten stay in the last branch of UNION ALL and access the base data instead.

Hint for Queries with Extended GROUP BY

You can use the EXPAND_GSET_TO_UNION hint to force expansion of the query with GROUP BY extensions into the equivalent UNION ALL query. This hint can be used in an environment where materialized views have simple GROUP BY clauses only. In this case, Oracle extends rewrite flexibility as each branch can be independently rewritten by a separate materialized view. See *Oracle Database Performance Tuning Guide* for more information regarding EXPAND_GSET_TO_UNION.

Query Rewrite in the Presence of Window Functions

Window functions are used to compute cumulative, moving and centered aggregates. These functions work with the following aggregates: SUM, AVG, MIN/MAX., COUNT, VARIANCE, STDDEV, FIRST_VALUE, and LAST_VALUE. A query with window function can be rewritten using exact text match rewrite. This requires that the materialized view definition also matches the query exactly. When there is no window function on the materialized view, then a query with a window function can be rewritten provided the aggregate in the query is found in the materialized view and all other eligibility checks such as the join computability checks are successful. A window function on the query is compared to the window function in the materialized view using its canonical form format. This enables query rewrite to rewrite even complex window functions.

When a query with a window function requires rollup during query rewrite, query rewrite will, whenever possible, split the query into an inner query with the aggregate and an outer query with the windowing function. This permits query rewrite to rewrite the aggregate in the inner query before applying the window function. One exception is when the query has both a window function and grouping sets. In this case, presence of the grouping set prevents query rewrite from splitting the query so query rewrite does not take place in this case.

Query Rewrite and Expression Matching

An expression that appears in a query can be replaced with a simple column in a materialized view provided the materialized view column represents a precomputed expression that matches with the expression in the query. If a query can be rewritten to use a materialized view, it will be faster. This is because materialized views contain precomputed calculations and do not need to perform expression computation.

The expression matching is done by first converting the expressions into canonical forms and then comparing them for equality. Therefore, two different expressions will generally be matched as long as they are equivalent to each other. Further, if the entire expression in a query fails to match with an expression in a materialized view, then subexpressions of it are tried to find a match. The subexpressions are tried in a top-down order to get maximal expression matching.

Consider a query that asks for sum of sales by age brackets (1-10, 11-20, 21-30, and so on).

```
CREATE MATERIALIZED VIEW sales_by_age_bracket_mv
ENABLE QUERY REWRITE AS
SELECT TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999) AS age_bracket,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, customers c WHERE s.cust_id=c.cust_id
```

```
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

The following query rewrites, using expression matching:

```
SELECT TO_CHAR(((2000-c.cust_year_of_birth)/10)-0.5,999), SUM(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id=c.cust_id
GROUP BY TO_CHAR((2000-c.cust_year_of_birth)/10-0.5,999);
```

This query is rewritten in terms of `sales_by_age_bracket_mv` based on the matching of the canonical forms of the age bracket expressions (that is, 2000 - `c.cust_year_of_birth`)/10-0.5), as follows:

```
SELECT age_bracket, sum_amount_sold FROM sales_by_age_bracket_mv;
```

Query Rewrite Using Partially Stale Materialized Views

When a partition of the detail table is updated, only specific sections of the materialized view are marked stale. The materialized view must have information that can identify the partition of the table corresponding to a particular row or group of the materialized view. The simplest scenario is when the partitioning key of the table is available in the `SELECT` list of the materialized view because this is the easiest way to map a row to a stale partition. The key points when using partially stale materialized views are:

- Query rewrite can use a materialized view in `ENFORCED` or `TRUSTED` mode if the rows from the materialized view used to answer the query are known to be `FRESH`.
- The fresh rows in the materialized view are identified by adding selection predicates to the materialized view's `WHERE` clause. Oracle rewrites a query with this materialized view if its answer is contained within this (restricted) materialized view.

The fact table `sales` is partitioned based on ranges of `time_id` as follows:

```
PARTITION BY RANGE (time_id)
(PARTITION SALES_Q1_1998
           VALUES LESS THAN (TO_DATE('01-APR-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q2_1998
           VALUES LESS THAN (TO_DATE('01-JUL-1998', 'DD-MON-YYYY')),
 PARTITION SALES_Q3_1998
           VALUES LESS THAN (TO_DATE('01-OCT-1998', 'DD-MON-YYYY')),
...
```

Suppose you have a materialized view grouping by `time_id` as follows:

```
CREATE MATERIALIZED VIEW sum_sales_per_city_mv
ENABLE QUERY REWRITE AS
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
GROUP BY time_id, prod_subcategory, cust_city;
```

Also suppose new data will be inserted for December 2000, which will be assigned to partition `sales_q4_2000`. For testing purposes, you can apply an arbitrary DML operation on `sales`, changing a different partition than `sales_q1_2000` as the following query requests data in this partition when this materialized view is fresh. For example, the following:

```
INSERT INTO SALES VALUES(17, 10, '01-DEC-2000', 4, 380, 123.45, 54321);
```

Until a refresh is done, the materialized view is generically stale and cannot be used for unlimited rewrite in enforced mode. However, because the table `sales` is partitioned and not all partitions have been modified, Oracle can identify all partitions that have not been touched. The optimizer can identify the fresh rows in the materialized view (the data which is unaffected by updates since the last refresh operation) by implicitly adding selection predicates to the materialized view defining query as follows:

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id < TO_DATE('01-OCT-2000','DD-MON-YYYY')
OR s.time_id >= TO_DATE('01-OCT-2001','DD-MON-YYYY'))
GROUP BY time_id, prod_subcategory, cust_city;
```

Note that the freshness of partially stale materialized views is tracked on a per-partition base, and not on a logical base. Because the partitioning strategy of the `sales` fact table is on a quarterly base, changes in December 2000 causes the complete partition `sales_q4_2000` to become stale.

Consider the following query, which asks for sales in quarters 1 and 2 of 2000:

```
SELECT s.time_id, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY')
GROUP BY time_id, prod_subcategory, cust_city;
```

Oracle Database knows that those ranges of rows in the materialized view are fresh and can therefore rewrite the query with the materialized view. The rewritten query looks as follows:

```
SELECT time_id, prod_subcategory, cust_city, sum_amount_sold
FROM sum_sales_per_city_mv
WHERE time_id BETWEEN TO_DATE('01-JAN-2000', 'DD-MON-YYYY')
AND TO_DATE('01-JUL-2000', 'DD-MON-YYYY');
```

Instead of the partitioning key, a partition marker (a function that identifies the partition given a rowid) can be present in the `SELECT` (and `GROUP BY` list) of the materialized view. You can use the materialized view to rewrite queries that require data from only certain partitions (identifiable by the partition-marker), for instance, queries that have a predicate specifying ranges of the partitioning keys containing entire partitions. See Chapter 10, "Advanced Materialized Views" for details regarding the supplied partition marker function `DBMS_MVIEW.PMARKER`.

The following example illustrates the use of a partition marker in the materialized view instead of directly using the partition key column:

```
CREATE MATERIALIZED VIEW sum_sales_per_city_2_mv
ENABLE QUERY REWRITE AS
SELECT DBMS_MVIEW.PMARKER(s.rowid) AS pmarker,
       t.fiscal_quarter_desc, p.prod_subcategory, c.cust_city,
       SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND   s.time_id = t.time_id
GROUP BY DBMS_MVIEW.PMARKER(s.rowid),
    p.prod_subcategory, c.cust_city, t.fiscal_quarter_desc;
```

Suppose you know that the partition sales_q1_2000 is fresh and DML changes have taken place for other partitions of the sales table. For testing purposes, you can apply an arbitrary DML operation on sales, changing a different partition than sales_q1_2000 when the materialized view is fresh. An example is the following:

```
INSERT INTO SALES VALUES(17, 10, '01-DEC-2000', 4, 380, 123.45, 54321);
```

Although the materialized view sum_sales_per_city_2_mv is now considered generically stale, Oracle Database can rewrite the following query using this materialized view. This query restricts the data to the partition sales_q1_2000, and selects only certain values of cust_city, as shown in the following:

```
SELECT p.prod_subcategory, c.cust_city, SUM(s.amount_sold) AS sum_amount_sold
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
AND c.cust_city= 'Nuernberg'
AND s.time_id >=TO_DATE('01-JAN-2000','dd-mon-yyyy')
AND s.time_id <  TO_DATE('01-APR-2000','dd-mon-yyyy')
GROUP BY prod_subcategory, cust_city;
```

Note that rewrite with a partially stale materialized view that contains a PMARKER function can only take place when the complete data content of one or more partitions is accessed and the predicate condition is on the partitioned fact table itself, as shown in the earlier example.

The DBMS_MVIEW.PMARKER function gives you exactly one distinct value for each partition. This dramatically reduces the number of rows in a potential materialized view compared to the partitioning key itself, but you are also giving up any detailed information about this key. The only information you know is the partition number and, therefore, the lower and upper boundary values. This is the trade-off for reducing the cardinality of the range partitioning column and thus the number of rows.

Assuming the value of p_marker for partition sales_q1_2000 is 31070, the previously shown queries can be rewritten against the materialized view as follows:

```
SELECT mv.prod_subcategory, mv.cust_city, SUM(mv.sum_amount_sold)
FROM sum_sales_per_city_2_mv mv
WHERE mv.pmarker = 31070 AND mv.cust_city= 'Nuernberg'
GROUP BY prod_subcategory, cust_city;
```

So the query can be rewritten against the materialized view without accessing stale data.

Cursor Sharing and Bind Variables

Query rewrite is supported when the query contains user bind variables as long as the actual bind values are not required during query rewrite. If the actual values of the bind variables are required during query rewrite, then we say that query rewrite is dependent on the bind values. Because the user bind variables are not available during query rewrite time, if query rewrite is dependent on the bind values, it is not possible to rewrite the query. For example, consider the following materialized view, customer_mv, which has the predicate, (customer_id >= 1000), in the WHERE clause:

```
CREATE MATERIALIZED VIEW customer_mv
ENABLE QUERY REWRITE AS
SELECT cust_id, prod_id,  SUM(amount_sold) AS total_amount
FROM sales WHERE cust_id >= 1000
GROUP BY cust_id, prod_id;
```

Consider the following query, which has a user bind variable, :user_id, in its WHERE clause:

```
SELECT cust_id, prod_id, SUM(amount_sold) AS sum_amount
FROM sales WHERE cust_id > :user_id
GROUP BY cust_id, prod_id;
```

Because the materialized view, customer_mv, has a selection in its WHERE clause, query rewrite is dependent on the actual value of the user bind variable, user_id, to compute the containment. Because user_id is not available during query rewrite time and query rewrite is dependent on the bind value of user_id, this query cannot be rewritten.

Even though the preceding example has a user bind variable in the WHERE clause, the same is true regardless of where the user bind variable appears in the query. In other words, irrespective of where a user bind variable appears in a query, if query rewrite is dependent on its value, then the query cannot be rewritten.

Now consider the following query which has a user bind variable, :user_id, in its SELECT list:

```
SELECT cust_id + :user_id, prod_id, SUM(amount_sold) AS total_amount
FROM sales WHERE cust_id >= 2000
GROUP BY cust_id, prod_id;
```

Because the value of the user bind variable, user_id, is not required during query rewrite time, the preceding query will rewrite.

```
SELECT cust_id + :user_id, prod_id, total_amount
FROM customer_mv;
```

Handling Expressions in Query Rewrite

Rewrite with some expressions is also supported when the expression evaluates to a constant, such as TO_DATE('12-SEP-1999','DD-Mon-YYYY'). For example, if an existing materialized view is defined as:

```
CREATE MATERIALIZED VIEW sales_on_valentines_day_99_mv
BUILD IMMEDIATE
REFRESH FORCE
ENABLE QUERY REWRITE AS
SELECT s.prod_id, s.cust_id, s.amount_sold
FROM times t, sales s WHERE s.time_id = t.time_id
AND t.time_id = TO_DATE('14-FEB-1999', 'DD-MON-YYYY');
```

Then the following query can be rewritten:

```
SELECT s.prod_id, s.cust_id, s.amount_sold
FROM sales s, times t WHERE s.time_id = t.time_id
AND t.time_id = TO_DATE('14-FEB-1999', 'DD-MON-YYYY');
```

This query would be rewritten as follows:

```
SELECT * FROM sales_on_valentines_day_99_mv;
```

Whenever TO_DATE is used, query rewrite only occurs if the date mask supplied is the same as the one specified by the NLS_DATE_FORMAT.

Advanced Query Rewrite Using Equivalences

There is a special type of query rewrite that is possible where a declaration is made that two SQL statements are functionally equivalent. This capability enables you to place inside application knowledge into the database so the database can exploit this knowledge for improved query performance. You do this by declaring two `SELECT` statements to be functionally equivalent (returning the same rows and columns) and indicating that one of the `SELECT` statements is more favorable for performance.

This advanced rewrite capability can generally be applied to a variety of query performance problems and opportunities. Any application can use this capability to affect rewrites against complex user queries that can be answered with much simpler and more performant queries that have been specifically created, usually by someone with inside application knowledge.

There are many scenarios where you can have inside application knowledge that would allow SQL statement transformation and tuning for significantly improved performance. The types of optimizations you may wish to affect can be very simple or as sophisticated as significant restructuring of the query. However, the incoming SQL queries are often generated by applications and you have no control over the form and structure of the application-generated queries.

To gain access to this capability, you need to connect as `SYSDBA` and explicitly grant execute access to the desired database administrators who will be declaring rewrite equivalences. See *Oracle Database PL/SQL Packages and Types Reference* for more information.

To illustrate this type of advanced rewrite, some examples using multidimensional data are provided. To optimize resource usage, an application may employ complicated SQL, custom C code or table functions to retrieve the data from the database. This complexity is irrelevant as far as end users are concerned. Users would still want to obtain their answers using typical queries with `SELECT ... GROUP BY`.

The following example declares to Oracle that a given user query must be executed using a specified alternative query. Oracle would recognize this relationship and every time the user asked the query, it would transparently rewrite it using the alternative. Thus, the user is saved from the trouble of understanding and writing SQL for complicated aggregate computations.

Example 19–11 Rewrite Using Equivalence

There are two base tables `sales_fact` and `geog_dim`. You can compute the total sales for each city, state and region with a rollup, by issuing the following statement:

```
SELECT g.region, g.state, g.city,
GROUPING_ID(g.city, g.state, g.region), SUM(sales)
FROM sales_fact f, geog_dim g WHERE f.geog_key = g.geog_key
GROUP BY ROLLUP(g.region, g.state, g.city);
```

An application may want to materialize this query for quick results. Unfortunately, the resulting materialized view occupies too much disk space. However, if you have a dimension rolling up city to state to region, you can easily compress the three grouping columns into one column using a decode statement. (This is also known as an embedded total):

```
DECODE (gid, 0, city, 1, state, 3, region, 7, "grand_total")
```

What this does is use the lowest level of the hierarchy to represent the entire information. For example, saying `Boston` means `Boston, MA, New England`

Region and saying CA means CA, Western Region. An application can store these embedded total results into a table, say, embedded_total_sales.

However, when returning the result back to the user, you would want to have all the data columns (city, state, region). In order to return the results efficiently and quickly, an application may use a custom table function (et_function) to retrieve the data back from the embedded_total_sales table in the expanded form as follows:

```
SELECT * FROM TABLE (et_function);
```

In other words, this feature allows an application to declare the equivalence of the user's preceding query to the alternative query, as in the following:

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'EMBEDDED_TOTAL',
   'SELECT g.region, g.state, g.city,
    GROUPING_ID(g.city, g.state, g.region), SUM(sales)
    FROM sales_fact f, geog_dim g
    WHERE f.geog_key = g.geog_key
    GROUP BY ROLLUP(g.region, g.state, g.city)',
    'SELECT * FROM TABLE(et_function)');
```

This invocation of DECLARE_REWRITE_EQUIVALENCE creates an equivalence declaration named EMBEDDED_TOTAL stating that the specified SOURCE_STMT and the specified DESTINATION_STMT are functionally equivalent, and that the specified DESTINATION_STMT is preferable for performance. After the DBA creates such a declaration, the user need have no knowledge of the space optimization being performed underneath the covers.

This capability also allows an application to perform specialized partial materializations of a SQL query. For instance, it could perform a rollup using a UNION ALL of three relations as shown in Example 19–12.

Example 19–12 Rewrite Using Equivalence (UNION ALL)

```
CREATE MATERIALIZED VIEW T1
AS SELECT g.region, g.state, g.city, 0 AS gid, SUM(sales) AS sales
FROM sales_fact f, geog_dim g WHERE f.geog_key = g.geog_key
GROUP BY g.region, g.state, g.city;

CREATE MATERIALIZED VIEW T2 AS
SELECT t.region, t.state, SUM(t.sales) AS sales
FROM T1 GROUP BY t.region, t.state;

CREATE VIEW T3 AS
SELECT t.region, SUM(t.sales) AS sales
FROM T2 GROUP BY t.region;
```

The ROLLUP(region, state, city) query is then equivalent to:

```
SELECT * FROM T1 UNION ALL
SELECT region, state, NULL, 1 AS gid, sales FROM T2 UNION ALL
SELECT region, NULL, NULL, 3 AS gid, sales FROM T3 UNION ALL
SELECT NULL, NULL, NULL, 7 AS gid, SUM(sales) FROM T3;
```

By specifying this equivalence, Oracle Database would use the more efficient second form of the query to compute the ROLLUP query asked by the user.

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'CUSTOM_ROLLUP',
   'SELECT g.region, g.state, g.city,
```

```
   GROUPING_ID(g.city, g.state, g.region), SUM(sales)
   FROM sales_fact f, geog_dim g
   WHERE f.geog_key = g.geog_key
   GROUP BY ROLLUP(g.region, g.state, g.city ',
   ' SELECT * FROM T1
   UNION ALL
   SELECT region, state, NULL, 1 as gid, sales FROM T2
   UNION ALL
   SELECT region, NULL, NULL, 3 as gid, sales FROM T3
   UNION ALL
   SELECT NULL, NULL, NULL, 7 as gid, SUM(sales) FROM T3');
```

Another application of this feature is to provide users special aggregate computations that may be conceptually simple but extremely complex to express in SQL. In this case, the application asks the user to use a specified custom aggregate function and internally compute it using complex SQL.

Example 19–13 Rewrite Using Equivalence (Using a Custom Aggregate)

Suppose the application users want to see the sales for each city, state and region and also additional sales information for specific seasons. For example, the New England user wants additional sales information for cities in New England for the winter months. The application would provide you a special aggregate `Seasonal_Agg` that computes the earlier aggregate. You would ask a classic summary query but use `Seasonal_Agg(sales, region)` rather than `SUM(sales)`.

```
SELECT g.region, t.calendar_month_name, Seasonal_Agg(f.sales, g.region) AS sales
FROM sales_fact f, geog_dim g, times t
WHERE f.geog_key = g.geog_key AND f.time_id = t.time_id
GROUP BY g.region, t.calendar_month_name;
```

Instead of asking the user to write SQL that does the extra computation, the application can do it automatically for them by using this feature. In this example, `Seasonal_Agg` is computed using the spreadsheet functionality (see Chapter 23, "SQL for Modeling"). Note that even though `Seasonal_Agg` is a user-defined aggregate, the required behavior is to add extra rows to the query's answer, which cannot be easily done with simple PL/SQL functions.

```
DBMS_ADVANCED_REWRITE.DECLARE_REWRITE_EQUIVALENCE (
   'CUSTOM_SEASONAL_AGG',
   SELECT g.region, t.calendar_month_name, Seasonal_Agg(sales, region) AS sales
   FROM sales_fact f, geog_dim g, times t
   WHERE f.geog_key = g.geog_key AND f.time_id = t.time_id
   GROUP BY g.region, t.calendar_month_name',
   'SELECT g,region, t.calendar_month_name, SUM(sales) AS sales
   FROM sales_fact f, geog_dim g
   WHERE f.geog_key = g.geog_key AND t.time_id = f.time_id
   GROUP BY g.region, g.state, g.city, t.calendar_month_name
   DIMENSION BY g.region, t.calendar_month_name
   (sales ['New England', 'Winter'] = AVG(sales) OVER calendar_month_name IN
    ('Dec', 'Jan', 'Feb', 'Mar'),
   sales ['Western', 'Summer' ] = AVG(sales) OVER calendar_month_name IN
   ('May', 'Jun', 'July', 'Aug'), .);
```

Creating Result Cache Materialized Views with Equivalences

A special type of materialized view, called a result cache materialized view (RCMV), enables you to use a result cache when running query rewrite. These result cache materialized views offer the main advantages of the result cache, faster access with

less space required, without the normal drawback of being unable to run query rewrite against them.

An example of using this type of materialized view is the following.

Example 19–14 Result Cache Materialized View

First, we grant the requisite permissions.

```
CONNECT / AS SYSDBA
GRANT CREATE MATERIALIZED VIEW TO sh;
GRANT EXECUTE ON DBMS_ADVANCED_REWRITE TO sh;
```

Next, we create the result cache materialized view.

```
CONNECT sh/sh
begin
   sys.DBMS_ADVANCED_REWRITE.Declare_Rewrite_Equivalence
    (
     Name               => 'RCMV_SALES',
     Source_Stmt        =>
      'select channel_id, prod_id, sum(amount_sold), count(amount_sold)
       from sales
       group by prod_id, channel_id',
     Destination_Stmt   =>
      'select * from
      (select /*+ RESULT_CACHE(name=RCMV_SALES) */
               channel_id, prod_id, sum(amount_sold), count(amount_sold)
         from sales
         group by prod_id, channel_id)',
       Validate         => FALSE,
       Rewrite_Mode     => 'GENERAL'
       );
end;
/

ALTER SESSION SET query_rewrite_integrity = stale_tolerated;
```

Then, we verify that different queries all rewrite to `RCMV_SALES` by looking at the explain plan.

```
EXPLAIN PLAN FOR
  SELECT channel_id, SUM(amount_sold) FROM sales GROUP BY channel_id;
@?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
Plan hash value: 3903632134
------------------------------------------------------------------------------------
|Id |         Operation       | Name  |Rows|Bytes|Cost(%CPU)| Time |Pstart|Pstop|
------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT        |       |  4 |   64| 1340 (68)|00:00:17|    |    |
| 1 |  HASH GROUP BY          |       |  4 |   64| 1340 (68)|00:00:17|    |    |
| 2 |   VIEW                  |       | 204| 3264| 1340 (68)|00:00:17|    |    |
| 3 |    RESULT CACHE         |3gps5zr86gyb53y36js9zuay2s| | | |     |    |    |
| 4 |     HASH GROUP BY       |       | 204| 2448| 1340 (68)|00:00:17|    |    |
| 5 |      PARTITION RANGE ALL|       |918K|  10M|  655 (33)|00:00:08|  1 | 28 |
| 6 |       TABLE ACCESS FULL | SALES |918K|  10M|  655 (33)|00:00:08|  1 | 28 |
------------------------------------------------------------------------------------

Result Cache Information (identified by operation id):
------------------------------------------------------
```

```
   3 - column-count=4; dependencies=(SH.SALES); name="RCMV_SALES"

18 rows selected.
```

Then, we execute the query that creates the cached result.

```
SELECT channel_id, SUM(amount_sold)
FROM sales
GROUP BY channel_id;

CHANNEL_ID  SUM(AMOUNT_SOLD)
----------  ----------------
         2        26346342.3
         4          13706802
         3        57875260.6
         9         277426.26
```

Next, we verify that the materialized view was materialized in the result cache.

```
CONNECT / AS SYSDBA

SELECT name, scan_count hits, block_count blocks, depend_count dependencies
FROM V$RESULT_CACHE_OBJECTS
WHERE name = 'RCMV_SALES';

NAME         HITS    BLOCKS   DEPENDENCIES
----------   ----    ------   ------------
RCMV_SALES      0         5              1
```

Finally, we drop the RCMV query equivalence.

```
begin
  sys.DBMS_ADVANCED_REWRITE.Drop_Rewrite_equivalence('RCMV_SALES');
end;
/
```

For more information regarding result caches, see *Oracle Database Performance Tuning Guide*.

Verifying that Query Rewrite has Occurred

Because query rewrite occurs transparently, special steps have to be taken to verify that a query has been rewritten. Of course, if the query runs faster, this should indicate that rewrite has occurred, but that is not proof. Therefore, to confirm that query rewrite does occur, use the `EXPLAIN PLAN` statement or the `DBMS_MVIEW.EXPLAIN_REWRITE` procedure.

Using EXPLAIN PLAN with Query Rewrite

The `EXPLAIN PLAN` facility is used as described in *Oracle Database SQL Language Reference*. For query rewrite, all you need to check is that the operation shows `MAT_VIEW REWRITE ACCESS`. If it does, then query rewrite has occurred. An example is the following, which creates the materialized view `cal_month_sales_mv`:

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT  t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

If EXPLAIN PLAN is used on the following SQL statement, the results are placed in the default table PLAN_TABLE. However, PLAN_TABLE must first be created using the utlxplan.sql script. Note that EXPLAIN PLAN does not actually execute the query.

```
EXPLAIN PLAN FOR
SELECT  t.calendar_month_desc, SUM(s.amount_sold)
FROM  sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

For the purposes of query rewrite, the only information of interest from PLAN_TABLE is the operation OBJECT_NAME, which identifies the method used to execute this query. Therefore, you would expect to see the operation MAT_VIEW REWRITE ACCESS in the output as illustrated in the following:

```
SELECT OPERATION, OBJECT_NAME FROM PLAN_TABLE;

OPERATION                 OBJECT_NAME
--------------------      ----------------------
SELECT STATEMENT
MAT_VIEW REWRITE ACCESS    CALENDAR_MONTH_SALES_MV
```

Using the EXPLAIN_REWRITE Procedure with Query Rewrite

It can be difficult to understand why a query did not rewrite. The rules governing query rewrite eligibility are quite complex, involving various factors such as constraints, dimensions, query rewrite integrity modes, freshness of the materialized views, and the types of queries themselves. In addition, you may want to know why query rewrite chose a particular materialized view instead of another. To help with this matter, Oracle provides the DBMS_MVIEW.EXPLAIN_REWRITE procedure to advise you when a query can be rewritten and, if not, why not. Using the results from DBMS_MVIEW.EXPLAIN_REWRITE, you can take the appropriate action needed to make a query rewrite if at all possible.

Note that the query specified in the EXPLAIN_REWRITE statement does not actually execute.

DBMS_MVIEW.EXPLAIN_REWRITE Syntax

You can obtain the output from DBMS_MVIEW.EXPLAIN_REWRITE in two ways. The first is to use a table, while the second is to create a VARRAY. The following shows the basic syntax for using an output table:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           VARCHAR2,
    mv              VARCHAR2(30),
    statement_id    VARCHAR2(30));
```

You can create an output table called REWRITE_TABLE by executing the utlxrw.sql script.

The query parameter is a text string representing the SQL query. The parameter, mv, is a fully-qualified materialized view name in the form of schema.mv. This is an optional parameter. When it is not specified, EXPLAIN_REWRITE returns any relevant messages regarding all the materialized views considered for rewriting the given query. When schema is omitted and only mv is specified, EXPLAIN_REWRITE looks for the materialized view in the current schema.

If you want to direct the output of EXPLAIN_REWRITE to a varray instead of a table, you should call the procedure as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE (
    query           [VARCHAR2 | CLOB],
    mv               VARCHAR2(30),
    output_array     SYS.RewriteArrayType);
```

Note that if the query is less than 256 characters long, EXPLAIN_REWRITE can be easily invoked with the EXECUTE command from SQL*Plus. Otherwise, the recommended method is to use a PL/SQL BEGIN... END block, as shown in the examples in /rdbms/demo/smxrw*.

Using REWRITE_TABLE

The output of EXPLAIN_REWRITE can be directed to a table named REWRITE_TABLE. You can create this output table by running the utlxrw.sql script. This script can be found in the admin directory. The format of REWRITE_TABLE is as follows:

```
CREATE TABLE REWRITE_TABLE(
  statement_id          VARCHAR2(30),   -- id for the query
  mv_owner              VARCHAR2(30),   -- owner of the MV
  mv_name               VARCHAR2(30),   -- name of the MV
  sequence              INTEGER,        -- sequence no of the msg
  query                 VARCHAR2(2000), -- user query
  query_block_no        INTEGER,        -- block no of the current subquery
  rewritten_txt         VARCHAR2(2000), -- rewritten query
  message               VARCHAR2(512),  -- EXPLAIN_REWRITE msg
  pass                  VARCHAR2(3),    -- rewrite pass no
  mv_in_msg             VARCHAR2(30),   -- MV in current message
  measure_in_msg        VARCHAR2(30),   -- Measure in current message
  join_back_tbl         VARCHAR2(30),   -- Join back table in message
  join_back_col         VARCHAR2(30),   -- Join back column in message
  original_cost         INTEGER,        -- Cost of original query
  rewritten_cost        INTEGER,        -- Cost of rewritten query
  flags                 INTEGER,        -- associated flags
  reserved1             INTEGER,        -- currently not used
  reerved2              VARCHAR2(10))   -- currently not used
;
```

Example 19–15 EXPLAIN_REWRITE Using REWRITE_TABLE

An example PL/SQL invocation is:

```
EXECUTE DBMS_MVIEW.EXPLAIN_REWRITE -
('SELECT p.prod_name, SUM(amount_sold) ' || -
'FROM sales s, products p ' || -
'WHERE s.prod_id = p.prod_id ' || -
' AND prod_name > ''B%'' ' || -
' AND prod_name < ''C%'' ' || -
'GROUP BY prod_name', -
'TestXRW.PRODUCT_SALES_MV', -
'SH');

SELECT message FROM rewrite_table ORDER BY sequence;
MESSAGE
--------------------------------------------------------------------------------
QSM-01033: query rewritten with materialized view, PRODUCT_SALES_MV
1 row selected.
```

The demo file xrwutl.sql contains a procedure that you can call to provide a more detailed output from EXPLAIN_REWRITE. See "EXPLAIN_REWRITE Output" on page 19-66 for more information.

The following is an example where you can see a more detailed explanation of why some materialized views were not considered and, eventually, the materialized view sales_mv was chosen as the best one.

```
DECLARE
  qrytext VARCHAR2(500)  :='SELECT cust_first_name, cust_last_name,
 SUM(amount_sold) AS dollar_sales FROM sales s, customers c WHERE s.cust_id=
 c.cust_id GROUP BY cust_first_name, cust_last_name';
    idno    VARCHAR2(30) :='ID1';
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE(qrytext, '', idno);
END;
/
SELECT message FROM rewrite_table ORDER BY sequence;
```

```
SQL> MESSAGE
--------------------------------------------------------------------------------
QSM-01082: Joining materialized view, CAL_MONTH_SALES_MV, with table, SALES, not possible
QSM-01022: a more optimal materialized view than PRODUCT_SALES_MV was used to rewrite
QSM-01022: a more optimal materialized view than FWEEK_PSCAT_SALES_MV was used to rewrite
QSM-01033: query rewritten with materialized view, SALES_MV
```

Using a Varray

You can save the output of EXPLAIN_REWRITE in a PL/SQL VARRAY. The elements of this array are of the type RewriteMessage, which is predefined in the SYS schema as shown in the following:

```
TYPE RewriteMessage IS OBJECT(
  mv_owner        VARCHAR2(30),   -- MV's schema
  mv_name         VARCHAR2(30),   -- Name of the MV
  sequence        NUMBER(3),      -- sequence no of the msg
  query_text      VARCHAR2(2000), -- User query
  query_block_no  NUMBER(3),      -- block no of the current subquery
  rewritten_text  VARCHAR2(2000), -- rewritten query text
  message         VARCHAR2(512),  -- EXPLAIN_REWRITE error msg
  pass            VARCHAR2(3),    -- Query rewrite pass no
  mv_in_msg       VARCHAR2(30),   -- MV in current message
  measure_in_msg  VARCHAR2(30),   -- Measure in current message
  join_back_tbl   VARCHAR2(30),   -- Join back table in current msg
  join_back_col   VARCHAR2(30),   -- Join back column in current msg
  original_cost   NUMBER(10),     -- Cost of original query
  rewritten_cost  NUMBER(10),     -- Cost rewritten query
  flags           NUMBER,         -- Associated flags
  reserved1       NUMBER,         -- For future use
  reserved2       VARCHAR2(10)    -- For future use
);
```

The array type, RewriteArrayType, which is a varray of RewriteMessage objects, is predefined in the SYS schema as follows:

- TYPE RewriteArrayType AS VARRAY(256) OF RewriteMessage;
- Using this array type, now you can declare an array variable and specify it in the EXPLAIN_REWRITE statement.
- Each RewriteMessage record provides a message concerning rewrite processing.
- The parameters are the same as for REWRITE_TABLE, except for statement_id, which is not used when using a varray as output.

- The mv_owner field defines the owner of materialized view that is relevant to the message.
- The mv_name field defines the name of a materialized view that is relevant to the message.
- The sequence field defines the sequence in which messages should be ordered.
- The query_text field contains the first 2000 characters of the query text under analysis.
- The message field contains the text of message relevant to rewrite processing of query.
- The flags, reserved1, and reserved2 fields are reserved for future use.

Example 19–16 EXPLAIN_REWRITE Using a VARRAY

Consider the following materialized view:

```
CREATE MATERIALIZED VIEW avg_sales_city_state_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_city, c.cust_state_province, AVG(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id = c.cust_id
GROUP BY c.cust_city, c.cust_state_province;
```

We might try to use this materialized view with the following query:

```
SELECT c.cust_state_province, AVG(s.amount_sold)
FROM sales s, customers c WHERE s.cust_id = c.cust_id
GROUP BY c.cust_state_province;
```

However, the query does not rewrite with this materialized view. This can be quite confusing to a novice user as it seems like all information required for rewrite is present in the materialized view. You can find out from DBMS_MVIEW.EXPLAIN_REWRITE that AVG cannot be computed from the given materialized view. The problem is that a ROLLUP is required here and AVG requires a COUNT or a SUM to do ROLLUP.

An example PL/SQL block for the previous query, using a VARRAY as its output, is as follows:

```
SET SERVEROUTPUT ON
DECLARE
  Rewrite_Array SYS.RewriteArrayType := SYS.RewriteArrayType();
  querytxt VARCHAR2(1500) := 'SELECT c.cust_state_province,
  AVG(s.amount_sold)
   FROM sales s, customers c WHERE s.cust_id = c.cust_id
   GROUP BY c.cust_state_province';
  i NUMBER;
BEGIN
  DBMS_MVIEW.EXPLAIN_REWRITE(querytxt, 'AVG_SALES_CITY_STATE_MV',
  Rewrite_Array);
  FOR i IN 1..Rewrite_Array.count
  LOOP
    DBMS_OUTPUT.PUT_LINE(Rewrite_Array(i).message);
  END LOOP;
END;
/
```

The following is the output of this EXPLAIN_REWRITE statement:

```
QSM-01065: materialized view, AVG_SALES_CITY_STATE_MV, cannot compute
```

```
  measure, AVG, in the query
QSM-01101: rollup(s) took place on mv, AVG_SALES_CITY_STATE_MV
QSM-01053: NORELY referential integrity constraint on table, CUSTOMERS,
  in TRUSTED/STALE TOLERATED integrity mode
PL/SQL procedure successfully completed.
```

EXPLAIN_REWRITE Benefit Statistics

The output of EXPLAIN_REWRITE contains two columns, original_cost and rewritten_cost, that can help you estimate query cost. original_cost gives the optimizer's estimation for the query cost when query rewrite was disabled. rewritten_cost gives the optimizer's estimation for the query cost when query was rewritten using a materialized view. These cost values can be used to find out what benefit a particular query receives from rewrite.

Support for Query Text Larger than 32KB in EXPLAIN_REWRITE

In this release, the EXPLAIN_REWRITE procedure has been enhanced to support large queries. The input query text can now be defined using a CLOB data type instead of a VARCHAR data type. This allows EXPLAIN_REWRITE to accept queries up to 4 GB.

The syntax for using EXPLAIN_REWRITE using CLOB to obtain the output into a table is shown as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(
   query          IN CLOB,
   mv             IN VARCHAR2,
   statement_id   IN VARCHAR2);
```

The second argument, mv, and the third argument, statement_id, can be NULL. Similarly, the syntax for using EXPLAIN_REWRITE using CLOB to obtain the output into a varray is shown as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(
   query           IN CLOB,
   mv              IN VARCHAR2,
   msg_array       IN OUT SYS.RewriteArrayType);
```

As before, the second argument, mv, can be NULL. Note that long query texts in CLOB can be generated using the procedures provided in the DBMS_LOB package.

EXPLAIN_REWRITE and Multiple Materialized Views

The syntax for using EXPLAIN_REWRITE with multiple materialized views is the same as using it with a single materialized view, except that the materialized views are specified by a comma-delimited string. For example, to find out whether a given set of materialized views mv1, mv2, and mv3 could be used to rewrite the query, query_txt, and, if not, why not, use EXPLAIN_REWRITE as follows:

```
DBMS_MVIEW.EXPLAIN_REWRITE(query_txt, 'mv1, mv2, mv3')
```

If the query, query_txt, rewrote with the given set of materialized views, then the following message appears:

```
QSM-01127: query rewritten with materialized view(s), mv1, mv2, and mv3.
```

If the query fails to rewrite with one or more of the given set of materialized views, then the reason for the failure will be output by EXPLAIN_REWRITE for each of the materialized views that did not participate in the rewrite.

EXPLAIN_REWRITE Output

Some examples showing how to use EXPLAIN_REWRITE are included in /rdbms/demo/smxrw.sql. There is also a utility called SYS.XRW included in the demo xrw area to help you select the output from the EXPLAIN_REWRITE procedure. When EXPLAIN_REWRITE evaluates a query, its output includes information such as the rewritten query text, query block number, and the cost of the rewritten query. The utility SYS.XRW outputs the user specified fields in a neatly formatted way, so that the output can be easily understood. The syntax is as follows:

```
SYS.XRW(list_of_mvs, list_of_commands, query_text),
```

where list_of_mvs are the materialized views the user would expect the query rewrite to use. If there is more than one materialized view, they must be separated by commas, and list_of_commands is one of the following fields:

```
QUERY_TXT:      User query text
REWRITTEN_TXT:  Rewritten query text
QUERY_BLOCK_NO: Query block number to identify each query blocks in
                case the query has subqueries or inline views
PASS:           Pass indicates whether a given message was generated
                before or after the view merging process of query rewrite.
COSTS:          Costs indicates the estimated execution cost of the
                original query and the rewritten query
```

The following example illustrates the use of this utility:

```
DROP MATERIALIZED VIEW month_sales_mv;

CREATE MATERIALIZED VIEW month_sales_mv
  ENABLE QUERY REWRITE
  AS
  SELECT t.calendar_month_number, SUM(s.amount_sold) AS sum_dollars
  FROM sales s, times t
  WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_number;

SET SERVEROUTPUT ON
DECLARE
  querytxt VARCHAR2(1500) := 'SELECT t.calendar_month_number,
         SUM(s.amount_sold) AS sum_dollars FROM sales s, times t
  WHERE s.time_id = t.time_id GROUP BY t.calendar_month_number';
BEGIN
  SYS.XRW('MONTH_SALES_MV', 'COSTS, PASS, REWRITTEN_TXT, QUERY_BLOCK_NO',
querytxt);
END;
/
```

Following is the output from SYS.XRW. As can be seen from the output, SYS.XRW outputs both the original query cost, rewritten costs, rewritten query text, query block number and whether the message was generated before or after the view merging process.

```
================================================================================
>> MESSAGE  : QSM-01151: query was rewritten
>> RW QUERY : SELECT MONTH_SALES_MV.CALENDAR_MONTH_NUMBER CALENDAR_MONTH_NUMBER,
MONTH_SALES_MV.SUM_DOLLARS SUM_DOLLARS FROM SH.MONTH_SALES_MV MONTH_SALES_MV
>> ORIG COST: 19.952763130792                  RW COST: 1.80687108
================================================================================
>>
------------------------ ANALYSIS OF QUERY REWRITE ------------------------
```

```
>>
>> QRY BLK #: 0
>> MESSAGE  : QSM-01209: query rewritten with materialized view,
   MONTH_SALES_MV, using text match algorithm
>> RW QUERY : SELECT MONTH_SALES_MV.CALENDAR_MONTH_NUMBER CALENDAR_MONTH_NUMBER,
   MONTH_SALES_MV.SUM_DOLLARS SUM_DOLLARS FROM SH.MONTH_SALES_MV MONTH_SALES_MV
>> ORIG COST: 19.952763130792                  RW COST: 1.80687108
>> MESSAGE OUTPUT BEFORE VIEW MERGING...
============================ END OF MESSAGES ==============================
PL/SQL procedure successfully completed.
```

Design Considerations for Improving Query Rewrite Capabilities

This section discusses design considerations that will help in obtaining the maximum benefit from query rewrite. They are not mandatory for using query rewrite and rewrite is not guaranteed if you follow them. They are general rules to consider, and are the following:

- Query Rewrite Considerations: Constraints
- Query Rewrite Considerations: Dimensions
- Query Rewrite Considerations: Outer Joins
- Query Rewrite Considerations: Text Match
- Query Rewrite Considerations: Aggregates
- Query Rewrite Considerations: Grouping Conditions
- Query Rewrite Considerations: Expression Matching
- Query Rewrite Considerations: Date Folding
- Query Rewrite Considerations: Statistics
- Query Rewrite Considerations: Hints

Query Rewrite Considerations: Constraints

Make sure all inner joins referred to in a materialized view have referential integrity (foreign key/primary key constraints) with additional `NOT NULL` constraints on the foreign key columns. Since constraints tend to impose a large overhead, you could make them `NO VALIDATE` and `RELY` and set the parameter `QUERY_REWRITE_INTEGRITY` to `STALE_TOLERATED` or `TRUSTED`. However, if you set `QUERY_REWRITE_INTEGRITY` to `ENFORCED`, all constraints must be enabled, enforced, and validated to get maximum rewritability.

You should avoid using the `ON DELETE` clause as it can lead to unexpected results.

Query Rewrite Considerations: Dimensions

You can express the hierarchical relationships and functional dependencies in normalized or denormalized dimension tables using the `HIERARCHY` and `DETERMINES` clauses of a dimension. Dimensions can express intra-table relationships which cannot be expressed by constraints. Set the parameter `QUERY_REWRITE_INTEGRITY` to `TRUSTED` or `STALE_TOLERATED` for query rewrite to take advantage of the relationships declared in dimensions.

Query Rewrite Considerations: Outer Joins

Another way of avoiding constraints is to use outer joins in the materialized view. Query rewrite will be able to derive an inner join in the query, such as `(A.a=B.b)`, from an outer join in the materialized view `(A.a = B.b(+))`, as long as the rowid of `B` or column `B.b` is available in the materialized view. Most of the support for rewrites with outer joins is provided for materialized views with joins only. To exploit it, a materialized view with outer joins should store the rowid or primary key of the inner table of an outer join. For example, the materialized view `join_sales_time_product_mv_oj` stores the primary keys `prod_id` and `time_id` of the inner tables of outer joins.

Query Rewrite Considerations: Text Match

If you need to speed up an extremely complex, long-running query, you could create a materialized view with the exact text of the query. Then the materialized view would contain the query results, thus eliminating the time required to perform any complex joins and search through all the data for that which is required.

Query Rewrite Considerations: Aggregates

To get the maximum benefit from query rewrite, make sure that all aggregates which are needed to compute ones in the targeted set of queries are present in the materialized view. The conditions on aggregates are quite similar to those for incremental refresh. For instance, if `AVG(x)` is in the query, then you should store `COUNT(x)` and `AVG(x)` or store `SUM(x)` and `COUNT(x)` in the materialized view. See "General Restrictions on Fast Refresh" on page 9-20 for fast refresh requirements.

Query Rewrite Considerations: Grouping Conditions

Aggregating data at lower levels in the hierarchy is better than aggregating at higher levels because lower levels can be used to rewrite more queries. Note, however, that doing so will also take up more space. For example, instead of grouping on state, group on city (unless space constraints prohibit it).

Instead of creating multiple materialized views with overlapping or hierarchically related `GROUP BY` columns, create a single materialized view with all those `GROUP BY` columns. For example, instead of using a materialized view that groups by city and another materialized view that groups by month, use a single materialized view that groups by city and month.

Use `GROUP BY` on columns that correspond to levels in a dimension but not on columns that are functionally dependent, because query rewrite will be able to use the functional dependencies automatically based on the `DETERMINES` clause in a dimension. For example, instead of grouping on `prod_name`, group on `prod_id` (as long as there is a dimension which indicates that the attribute `prod_id` determines `prod_name`, you will enable the rewrite of a query involving `prod_name`).

Query Rewrite Considerations: Expression Matching

If several queries share the same common subselect, it is advantageous to create a materialized view with the common subselect as one of its `SELECT` columns. This way, the performance benefit due to precomputation of the common subselect can be obtained across several queries.

Query Rewrite Considerations: Date Folding

When creating a materialized view that aggregates data by folded date granules such as months or quarters or years, always use the year component as the prefix but not as the suffix. For example, `TO_CHAR(date_col,'yyyy-q')` folds the date into quarters, which collate in year order, whereas `TO_CHAR(date_col, 'q-yyyy')` folds the date into quarters, which collate in quarter order. The former preserves the ordering while the latter does not. For this reason, any materialized view created without a year prefix will not be eligible for date folding rewrite.

Query Rewrite Considerations: Statistics

Optimization with materialized views is based on cost and the optimizer needs statistics of both the materialized view and the tables in the query to make a cost-based choice. Materialized views should thus have statistics collected using the `DBMS_STATS` package.

Query Rewrite Considerations: Hints

This section discusses the following considerations:

- REWRITE and NOREWRITE Hints
- REWRITE_OR_ERROR Hint
- Multiple Materialized View Rewrite Hints
- EXPAND_GSET_TO_UNION Hint

REWRITE and NOREWRITE Hints

You can include hints in the `SELECT` blocks of your SQL statements to control whether query rewrite occurs. Using the `NOREWRITE` hint in a query prevents the optimizer from rewriting it.

The `REWRITE` hint with no argument in a query forces the optimizer to use a materialized view (if any) to rewrite it regardless of the cost. If you use the `REWRITE(mv1,mv2,...)` hint with arguments, this forces rewrite to select the most suitable materialized view from the list of names specified.

To prevent a rewrite, you can use the following statement:

```
SELECT /*+ NOREWRITE */ p.prod_subcategory, SUM(s.amount_sold)
FROM   sales s, products p WHERE  s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

To force a rewrite using `sum_sales_pscat_week_mv` (if such a rewrite is possible), use the following statement:

```
SELECT /*+ REWRITE (sum_sales_pscat_week_mv) */
       p.prod_subcategory,  SUM(s.amount_sold)
FROM   sales s, products p WHERE  s.prod_id=p.prod_id
GROUP BY p.prod_subcategory;
```

Note that the scope of a rewrite hint is a query block. If a SQL statement consists of several query blocks (`SELECT` clauses), you must specify a rewrite hint on each query block to control the rewrite for the entire statement.

REWRITE_OR_ERROR Hint

Using the REWRITE_OR_ERROR hint in a query causes the following error if the query failed to rewrite:

```
ORA-30393: a query block in the statement did not rewrite
```

For example, the following query issues an ORA-30393 error when there are no suitable materialized views for query rewrite to use:

```
SELECT /*+ REWRITE_OR_ERROR */ p.prod_subcategory, SUM(s.amount_sold)
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_subcategory;
```

Multiple Materialized View Rewrite Hints

There are two hints to control rewrites when using multiple materialized views. The NO_MULTIMV_REWRITE hint prevents the query from being rewritten with more than one materialized view and the NO_BASETABLE_MULTIMV_REWRITE hint prevents the query from being rewritten with a combination of materialized views and the base tables.

EXPAND_GSET_TO_UNION Hint

You can use the EXPAND_GSET_TO_UNION hint to force expansion of the query with GROUP BY extensions into the equivalent UNION ALL query. See "Hint for Queries with Extended GROUP BY" on page 19-51 for further information.

20

Schema Modeling Techniques

The following topics provide information about schemas in a data warehouse:

- Schemas in Data Warehouses
- Third Normal Form
- Star Schemas
- Optimizing Star Queries

Schemas in Data Warehouses

A **schema** is a collection of database objects, including tables, views, indexes, and synonyms.

There is a variety of ways of arranging schema objects in the schema models designed for data warehousing. One data warehouse schema model is a star schema. The `sh` sample schema (the basis for most of the examples in this book) uses a star schema. However, there are other schema models that are commonly used for data warehouses. The most prevalent of these schema models is the **third normal form (3NF)** schema. Additionally, some data warehouse schemas are neither star schemas nor 3NF schemas, but instead share characteristics of both schemas; these are referred to as hybrid schema models.

The Oracle Database is designed to support all data warehouse schemas. Some features may be specific to one schema model (such as the star transformation feature, described in "Using Star Transformation" on page 20-4, which is specific to star schemas). However, the vast majority of Oracle's data warehousing features are equally applicable to star schemas, 3NF schemas, and hybrid schemas. Key data warehousing capabilities such as partitioning (including the rolling window load technique), parallelism, materialized views, and analytic SQL are implemented in all schema models.

The determination of which schema model should be used for a data warehouse should be based upon the requirements and preferences of the data warehouse project team. Comparing the merits of the alternative schema models is outside of the scope of this book; instead, this chapter briefly introduces each schema model and suggest how Oracle Database can be optimized for those environments.

Third Normal Form

Although this guide primarily uses star schemas in its examples, you can also use the third normal form for your data warehouse implementation.

Third normal form modeling is a classical relational-database modeling technique that minimizes data redundancy through normalization. When compared to a star schema, a 3NF schema typically has a larger number of tables due to this normalization process. For example, in Figure 20–1, `orders` and `order items` tables contain similar information as `sales` table in the star schema in Figure 20–2.

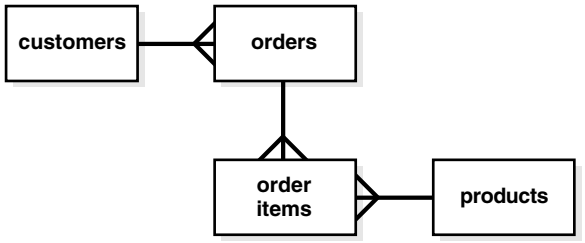
3NF schemas are typically chosen for large data warehouses, especially environments with significant data-loading requirements that are used to feed data marts and execute long-running queries.

The main advantages of 3NF schemas are that they:

- Provide a neutral schema design, independent of any application or data-usage considerations
- May require less data-transformation than more normalized schemas such as star schemas

Figure 20–1 presents a graphical representation of a third normal form schema.

Figure 20–1 Third Normal Form Schema



Optimizing Third Normal Form Queries

Queries on 3NF schemas are often very complex and involve a large number of tables. The performance of joins between large tables is thus a primary consideration when using 3NF schemas.

One particularly important feature for 3NF schemas is partition-wise joins. The largest tables in a 3NF schema should be partitioned to enable partition-wise joins. The most common partitioning technique in these environments is composite range-hash partitioning for the largest tables, with the most-common join key chosen as the hash-partitioning key.

Parallelism is often heavily utilized in 3NF environments, and parallelism should typically be enabled in these environments.

Star Schemas

The **star schema** is perhaps the simplest data warehouse schema. It is called a star schema because the entity-relationship diagram of this schema resembles a star, with points radiating from a central table. The center of the star consists of a large fact table and the points of the star are the dimension tables.

A **star query** is a join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other. The optimizer recognizes star queries and generates efficient execution plans for them. It is not mandatory to have any foreign keys on the fact table for star transformation to take effect.

A typical fact table contains keys and measures. For example, in the `sh` sample schema, the fact table, `sales`, contain the measures `quantity_sold`, `amount`, and `cost`, and the keys `cust_id`, `time_id`, `prod_id`, `channel_id`, and `promo_id`. The dimension tables are `customers`, `times`, `products`, `channels`, and `promotions`. The `products` dimension table, for example, contains information about each product number that appears in the fact table.

A star join is a primary key to foreign key join of the dimension tables to a fact table.

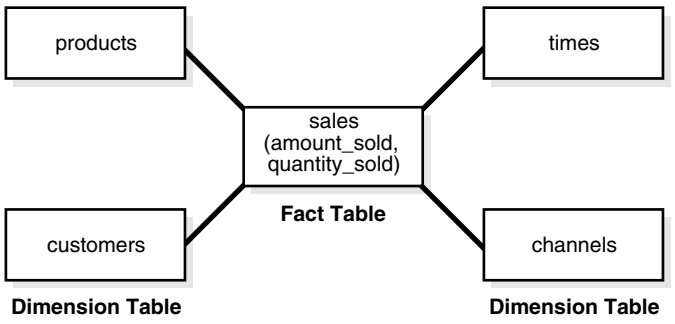
The main advantages of star schemas are that they:

- Provide a direct and intuitive mapping between the business entities being analyzed by end users and the schema design.
- Provide highly optimized performance for typical star queries.
- Are widely supported by a large number of business intelligence tools, which may anticipate or even require that the data warehouse schema contain dimension tables.

Star schemas are used for both simple data marts and very large data warehouses.

Figure 20–2 presents a graphical representation of a star schema.

Figure 20–2 Star Schema

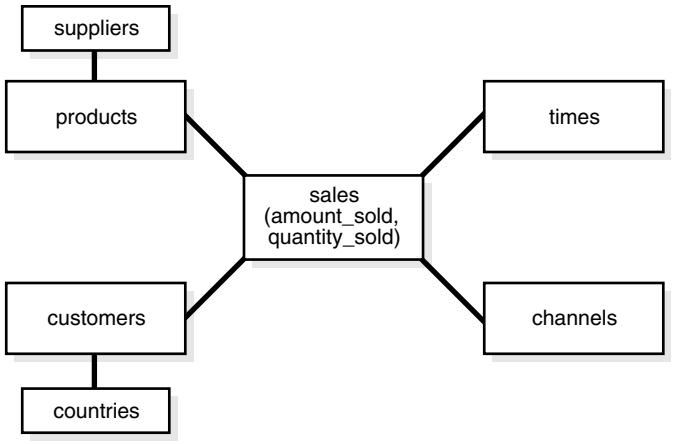


Snowflake Schemas

The snowflake schema is a more complex data warehouse model than a star schema, and is a type of star schema. It is called a snowflake schema because the diagram of the schema resembles a snowflake.

Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a products table, a `product_category` table, and a `product_manufacturer` table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance. Figure 20–3 presents a graphical representation of a snowflake schema.

Figure 20–3 Snowflake Schema



Note: Oracle recommends you choose a star schema over a snowflake schema unless you have a clear reason not to.

Optimizing Star Queries

You should consider the following when using star queries:

- Tuning Star Queries
- Using Star Transformation

Tuning Star Queries

To get the best possible performance for star queries, it is important to follow some basic guidelines:

- A bitmap index should be built on each of the foreign key columns of the fact table or tables.
- The initialization parameter `STAR_TRANSFORMATION_ENABLED` should be set to `TRUE`. This enables an important optimizer feature for star-queries. It is set to `FALSE` by default for backward-compatibility.

When a data warehouse satisfies these conditions, the majority of the star queries running in the data warehouse uses a query execution strategy known as the star transformation. The star transformation provides very efficient query performance for star queries.

Using Star Transformation

The star transformation is a powerful optimization technique that relies upon implicitly rewriting (or transforming) the SQL of the original star query. The end user never needs to know any of the details about the star transformation. Oracle Database's query optimizer automatically chooses the star transformation where appropriate.

The star transformation is a query transformation aimed at executing star queries efficiently. Oracle Database processes a star query using two basic phases. The first phase retrieves exactly the necessary rows from the fact table (the result set). Because this retrieval utilizes bitmap indexes, it is very efficient. The second phase joins this

result set to the dimension tables. An example of an end user query is: "What were the sales and profits for the grocery department of stores in the west and southwest sales districts over the last three quarters?" This is a simple star query.

Star Transformation with a Bitmap Index

A prerequisite of the star transformation is that there be a single-column bitmap index on every join column of the fact table. These join columns include all foreign key columns.

For example, the `sales` table of the `sh` sample schema has bitmap indexes on the `time_id`, `channel_id`, `cust_id`, `prod_id`, and `promo_id` columns.

Consider the following star query:

```
SELECT ch.channel_class, c.cust_city, t.calendar_quarter_desc,
   SUM(s.amount_sold) sales_amount
FROM sales s, times t, customers c, channels ch
WHERE s.time_id = t.time_id
AND   s.cust_id = c.cust_id
AND   s.channel_id = ch.channel_id
AND   c.cust_state_province = 'CA'
AND   ch.channel_desc in ('Internet','Catalog')
AND   t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY ch.channel_class, c.cust_city, t.calendar_quarter_desc;
```

This query is processed in two phases. In the first phase, Oracle Database uses the bitmap indexes on the foreign key columns of the fact table to identify and retrieve only the necessary rows from the fact table. That is, Oracle Database retrieves the result set from the fact table using essentially the following query:

```
SELECT ... FROM sales
WHERE time_id IN
  (SELECT time_id FROM times
   WHERE calendar_quarter_desc IN('1999-Q1','1999-Q2'))
   AND cust_id IN
  (SELECT cust_id FROM customers WHERE cust_state_province='CA')
   AND channel_id IN
  (SELECT channel_id FROM channels WHERE channel_desc IN('Internet','Catalog'));
```

This is the transformation step of the algorithm, because the original star query has been transformed into this subquery representation. This method of accessing the fact table leverages the strengths of bitmap indexes. Intuitively, bitmap indexes provide a set-based processing scheme within a relational database. Oracle has implemented very fast methods for doing set operations such as `AND` (an intersection in standard set-based terminology), `OR` (a set-based union), `MINUS`, and `COUNT`.

In this star query, a bitmap index on `time_id` is used to identify the set of all rows in the fact table corresponding to `sales` in `1999-Q1`. This set is represented as a bitmap (a string of 1's and 0's that indicates which rows of the fact table are members of the set).

A similar bitmap is retrieved for the fact table rows corresponding to the sale from `1999-Q2`. The bitmap `OR` operation is used to combine this set of `Q1` sales with the set of `Q2` sales.

Additional set operations will be done for the `customer` dimension and the `product` dimension. At this point in the star query processing, there are three bitmaps. Each bitmap corresponds to a separate dimension table, and each bitmap represents the set of rows of the fact table that satisfy that individual dimension's constraints.

These three bitmaps are combined into a single bitmap using the bitmap AND operation. This final bitmap represents the set of rows in the fact table that satisfy all of the constraints on the dimension table. This is the result set, the exact set of rows from the fact table needed to evaluate the query. Note that none of the actual data in the fact table has been accessed. All of these operations rely solely on the bitmap indexes and the dimension tables. Because of the bitmap indexes' compressed data representations, the bitmap set-based operations are extremely efficient.

Once the result set is identified, the bitmap is used to access the actual data from the sales table. Only those rows that are required for the end user's query are retrieved from the fact table. At this point, Oracle Database has effectively joined all of the dimension tables to the fact table using bitmap indexes. This technique provides excellent performance because Oracle Database is joining all of the dimension tables to the fact table with one logical join operation, rather than joining each dimension table to the fact table independently.

The second phase of this query is to join these rows from the fact table (the result set) to the dimension tables. Oracle uses the most efficient method for accessing and joining the dimension tables. Many dimension are very small, and table scans are typically the most efficient access method for these dimension tables. For large dimension tables, table scans may not be the most efficient access method. In the previous example, a bitmap index on `product.department` can be used to quickly identify all of those products in the grocery department. Oracle Database's optimizer automatically determines which access method is most appropriate for a given dimension table, based upon the optimizer's knowledge about the sizes and data distributions of each dimension table.

The specific join method (as well as indexing method) for each dimension table will likewise be intelligently determined by the optimizer. A hash join is often the most efficient algorithm for joining the dimension tables. The final answer is returned to the user once all of the dimension tables have been joined. The query technique of retrieving only the matching rows from one table and then joining to another table is commonly known as a semijoin.

Execution Plan for a Star Transformation with a Bitmap Index

The following typical execution plan might result from "Star Transformation with a Bitmap Index" on page 20-5:

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                          CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                         CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                        TIMES
     PARTITION RANGE ITERATOR
      TABLE ACCESS BY LOCAL INDEX ROWID       SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL                 CUSTOMERS
           BITMAP INDEX RANGE SCAN            SALES_CUST_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
```

```
          TABLE ACCESS FULL                CHANNELS
          BITMAP INDEX RANGE SCAN          SALES_CHANNEL_BIX
        BITMAP MERGE
         BITMAP KEY ITERATION
          BUFFER SORT
           TABLE ACCESS FULL               TIMES
          BITMAP INDEX RANGE SCAN          SALES_TIME_BIX
```

In this plan, the fact table is accessed through a bitmap access path based on a bitmap AND, of three merged bitmaps. The three bitmaps are generated by the BITMAP MERGE row source being fed bitmaps from row source trees underneath it. Each such row source tree consists of a BITMAP KEY ITERATION row source which fetches values from the subquery row source tree, which in this example is a full table access. For each such value, the BITMAP KEY ITERATION row source retrieves the bitmap from the bitmap index. After the relevant fact table rows have been retrieved using this access path, they are joined with the dimension tables and temporary tables to produce the answer to the query.

Star Transformation with a Bitmap Join Index

In addition to bitmap indexes, you can use a bitmap join index during star transformations. Assume you have the following additional index structure:

```
CREATE BITMAP INDEX sales_c_state_bjix
ON sales(customers.cust_state_province)
FROM sales, customers
WHERE sales.cust_id = customers.cust_id
LOCAL NOLOGGING COMPUTE STATISTICS;
```

The processing of the same star query using the bitmap join index is similar to the previous example. The only difference is that Oracle utilizes the join index, instead of a single-table bitmap index, to access the customer data in the first phase of the star query.

Execution Plan for a Star Transformation with a Bitmap Join Index

The following typical execution plan might result from "Execution Plan for a Star Transformation with a Bitmap Join Index" on page 20-7:

```
SELECT STATEMENT
 SORT GROUP BY
  HASH JOIN
   TABLE ACCESS FULL                       CHANNELS
   HASH JOIN
    TABLE ACCESS FULL                      CUSTOMERS
    HASH JOIN
     TABLE ACCESS FULL                     TIMES
     PARTITION RANGE ALL
      TABLE ACCESS BY LOCAL INDEX ROWID    SALES
       BITMAP CONVERSION TO ROWIDS
        BITMAP AND
         BITMAP INDEX SINGLE VALUE         SALES_C_STATE_BJIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
            TABLE ACCESS FULL              CHANNELS
           BITMAP INDEX RANGE SCAN         SALES_CHANNEL_BIX
         BITMAP MERGE
          BITMAP KEY ITERATION
           BUFFER SORT
```

```
 TABLE ACCESS FULL                TIMES
BITMAP INDEX RANGE SCAN           SALES_TIME_BIX
```

The difference between this plan as compared to the previous one is that the inner part of the bitmap index scan for the `customer` dimension has no subselect. This is because the join predicate information on `customer.cust_state_province` can be satisfied with the bitmap join index `sales_c_state_bjix`.

How Oracle Chooses to Use Star Transformation

The optimizer generates and saves the best plan it can produce without the transformation. If the transformation is enabled, the optimizer then tries to apply it to the query and, if applicable, generates the best plan using the transformed query. Based on a comparison of the cost estimates between the best plans for the two versions of the query, the optimizer then decides whether to use the best plan for the transformed or untransformed version.

If the query requires accessing a large percentage of the rows in the fact table, it might be better to use a full table scan and not use the transformations. However, if the constraining predicates on the dimension tables are sufficiently selective that only a small portion of the fact table must be retrieved, the plan based on the transformation will probably be superior.

Note that the optimizer generates a subquery for a dimension table only if it decides that it is reasonable to do so based on a number of criteria. There is no guarantee that subqueries will be generated for all dimension tables. The optimizer may also decide, based on the properties of the tables and the query, that the transformation does not merit being applied to a particular query. In this case the best regular plan will be used.

Star Transformation Restrictions

Star transformation is not supported for tables with any of the following characteristics:

- Queries with a table hint that is incompatible with a bitmap access path
- Queries that contain bind variables
- Tables with too few bitmap indexes. There must be a bitmap index on a fact table column for the optimizer to generate a subquery for it.
- Remote fact tables. However, remote dimension tables are allowed in the subqueries that are generated.
- Anti-joined tables
- Tables that are already used as a dimension table in a subquery
- Tables that are really unmerged views, which are not view partitions
- Tables where the fact table is an unmerged view.
- Tables where a partitioned view is used as a fact table.

The star transformation may not be chosen by the optimizer for the following cases:

- Tables that have a good single-table access path
- Tables that are too small for the transformation to be worthwhile

In addition, temporary tables will not be used by star transformation under the following conditions:

- The database is in read-only mode
- The star query is part of a transaction that is in serializable mode

21

SQL for Aggregation in Data Warehouses

This chapter discusses aggregation of SQL, a basic aspect of data warehousing. It contains these topics:

Overview of SQL for Aggregation in Data Warehouses

Aggregation is a fundamental part of data warehousing. To improve aggregation performance in your warehouse, Oracle Database provides the following functionality:

- `CUBE` and `ROLLUP` extensions to the `GROUP BY` clause
- Three `GROUPING` functions
- `GROUPING SETS` expression
- Pivoting operations

The `CUBE`, `ROLLUP`, and `GROUPING SETS` extensions to SQL make querying and reporting easier and faster. `CUBE`, `ROLLUP`, and grouping sets produce a single result set that is equivalent to a `UNION ALL` of differently grouped rows. `ROLLUP` calculates aggregations such as `SUM`, `COUNT`, `MAX`, `MIN`, and `AVG` at increasing levels of aggregation, from the most detailed up to a grand total. `CUBE` is an extension similar to `ROLLUP`, enabling a single statement to calculate all possible combinations of aggregations. The `CUBE`, `ROLLUP`, and the `GROUPING SETS` extension lets you specify just the groupings needed in the `GROUP BY` clause. This allows efficient analysis across multiple dimensions without performing a `CUBE` operation. Computing a `CUBE` creates a heavy processing load, so replacing cubes with grouping sets can significantly increase performance.

To enhance performance, CUBE, ROLLUP, and GROUPING SETS can be parallelized: multiple processes can simultaneously execute all of these statements. These capabilities make aggregate calculations more efficient, thereby enhancing database performance, and scalability.

The three GROUPING functions help you identify the group each row belongs to and enable sorting subtotal rows and filtering results.

Analyzing Across Multiple Dimensions

One of the key concepts in decision support systems is multidimensional analysis: examining the enterprise from all necessary combinations of dimensions. We use the term **dimension** to mean any category used in specifying questions. Among the most commonly specified dimensions are time, geography, product, department, and distribution channel, but the potential dimensions are as endless as the varieties of enterprise activity. The events or entities associated with a particular set of dimension values are usually referred to as facts. The facts might be sales in units or local currency, profits, customer counts, production volumes, or anything else worth tracking.

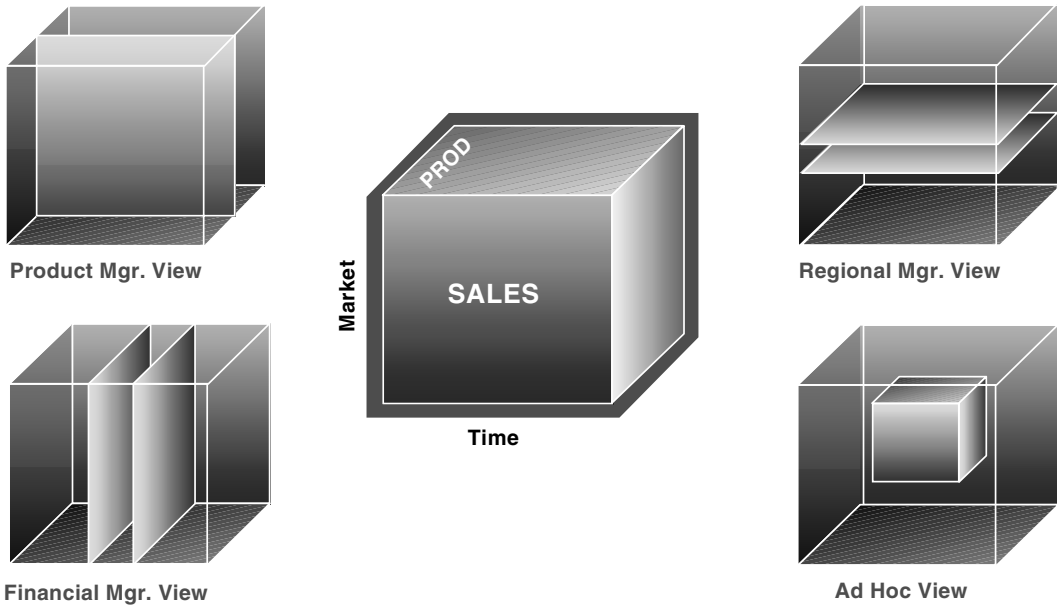
Here are some examples of multidimensional requests:

- Show total sales across all products at increasing aggregation levels for a geography dimension, from state to country to region, for 1999 and 2000.
- Create a cross-tabular analysis of our operations showing expenses by territory in South America for 1999 and 2000. Include all possible subtotals.
- List the top 10 sales representatives in Asia according to 2000 sales revenue for automotive products, and rank their commissions.

All these requests involve multiple dimensions. Many multidimensional questions require aggregated data and comparisons of data sets, often across time, geography or budgets.

To visualize data that has many dimensions, analysts commonly use the analogy of a data cube, that is, a space where facts are stored at the intersection of n dimensions. Figure 21–1 shows a data cube and how it can be used differently by various groups. The cube stores sales data organized by the dimensions of product, market, sales, and time. Note that this is only a metaphor: the actual data is physically stored in normal tables. The cube data consists of both detail and aggregated data.

Figure 21–1 Logical Cubes and Views by Different Users



You can retrieve slices of data from the cube. These correspond to cross-tabular reports such as the one shown in Table 21–1. Regional managers might study the data by comparing slices of the cube applicable to different markets. In contrast, product managers might compare slices that apply to different products. An ad hoc user might work with a wide variety of constraints, working in a subset cube.

Answering multidimensional questions often involves accessing and querying huge quantities of data, sometimes in millions of rows. Because the flood of detailed data generated by large organizations cannot be interpreted at the lowest level, aggregated views of the information are essential. Aggregations, such as sums and counts, across many dimensions are vital to multidimensional analyses. Therefore, analytical tasks require convenient and efficient data aggregation.

Optimized Performance

Not only multidimensional issues, but all types of processing can benefit from enhanced aggregation facilities. Transaction processing, financial and manufacturing systems—all of these generate large numbers of production reports needing substantial system resources. Improved efficiency when creating these reports will reduce system load. In fact, any computer process that aggregates data from details to higher levels will benefit from optimized aggregation performance.

These extensions provide aggregation features and bring many benefits, including:

- Simplified programming requiring less SQL code for many tasks.
- Quicker and more efficient query processing.
- Reduced client processing loads and network traffic because aggregation work is shifted to servers.
- Opportunities for caching aggregations because similar queries can leverage existing work.

An Aggregate Scenario

To illustrate the use of the GROUP BY extension, this chapter uses the sh data of the sample schema. All the examples refer to data from this scenario. The hypothetical company has sales across the world and tracks sales by both dollars and quantities information. Because there are many rows of data, the queries shown here typically have tight constraints on their WHERE clauses to limit the results to a small number of rows.

Example 21–1 Simple Cross-Tabular Report With Subtotals

Table 21–1 is a sample cross-tabular report showing the total sales by country_id and channel_desc for the US and France through the Internet and direct sales in September 2000.

Table 21–1 Simple Cross-Tabular Report With Subtotals

Channel	Country		
	France	US	Total
Internet	9,597	124,224	133,821
Direct Sales	61,202	638,201	699,403
Total	70,799	762,425	833,224

Consider that even a simple report such as this, with just nine values in its grid, generates four subtotals and a grand total. Half of the values needed for this report would not be calculated with a query that requested SUM(amount_sold) and did a GROUP BY(channel_desc, country_id). To get the higher-level aggregates would require additional queries. Database commands that offer improved calculation of subtotals bring major benefits to querying, reporting, and analytical operations.

```
SELECT channels.channel_desc, countries.country_iso_code,
  TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc='2000-09'
  AND customers.country_id=countries.country_id
  AND countries.country_iso_code IN ('US','FR')
GROUP BY CUBE(channels.channel_desc, countries.country_iso_code);

CHANNEL_DESC         CO SALES$
-------------------- -- --------------
                             833,224
                     FR       70,799
                     US      762,425
Internet                     133,821
Internet             FR        9,597
Internet             US      124,224
Direct Sales                 699,403
Direct Sales         FR       61,202
Direct Sales         US      638,201
```

Interpreting NULLs in Examples

NULLs returned by the GROUP BY extensions are not always the traditional null meaning value unknown. Instead, a NULL may indicate that its row is a subtotal. To

avoid introducing another non-value in the database system, these subtotal values are not given a special tag.

> **See Also:** "GROUPING Functions" on page 21-10 for details on how the nulls representing subtotals are distinguished from nulls stored in the data

ROLLUP Extension to GROUP BY

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.

The action of ROLLUP is straightforward: it creates subtotals that roll up from the most detailed level to a grand total, following a grouping list specified in the ROLLUP clause. ROLLUP takes as its argument an ordered list of grouping columns. First, it calculates the standard aggregate values specified in the GROUP BY clause. Then, it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. Finally, it creates a grand total.

ROLLUP creates subtotals at n+1 levels, where n is the number of grouping columns. For instance, if a query specifies ROLLUP on grouping columns of time, region, and department(n=3), the result set will include rows at four aggregation levels.

You might want to compress your data when using ROLLUP. This is particularly useful when there are few updates to older partitions.

When to Use ROLLUP

Use the ROLLUP extension in tasks involving subtotals.

- It is very helpful for subtotaling along a hierarchical dimension such as time or geography. For instance, a query could specify a ROLLUP(y, m, day) or ROLLUP(country, state, city).
- For data warehouse administrators using summary tables, ROLLUP can simplify and speed up the maintenance of summary tables.

ROLLUP Syntax

ROLLUP appears in the GROUP BY clause in a SELECT statement. Its form is:

```
SELECT … GROUP BY ROLLUP(grouping_column_reference_list)
```

Example 21–2 ROLLUP

This example uses the data in the sh sample schema data, the same data as was used in Figure 21–1. The ROLLUP is across three dimensions.

```
SELECT channels.channel_desc, calendar_month_desc,
       countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id
  AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id = channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
```

```
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY
  ROLLUP(channels.channel_desc, calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet             2000-09  GB         16,569
Internet             2000-09  US        124,224
Internet             2000-09            140,793
Internet             2000-10  GB         14,539
Internet             2000-10  US        137,054
Internet             2000-10            151,593
Internet                                292,387
Direct Sales         2000-09  GB         85,223
Direct Sales         2000-09  US        638,201
Direct Sales         2000-09            723,424
Direct Sales         2000-10  GB         91,925
Direct Sales         2000-10  US        682,297
Direct Sales         2000-10            774,222
Direct Sales                          1,497,646
                                      1,790,032
```

Note that results do not always add up due to rounding.

This query returns the following sets of rows:

- Regular aggregation rows that would be produced by `GROUP BY` without using `ROLLUP`.
- First-level subtotals aggregating across `country_id` for each combination of `channel_desc` and `calendar_month`.
- Second-level subtotals aggregating across `calendar_month_desc` and `country_id` for each `channel_desc` value.
- A grand total row.

Partial Rollup

You can also roll up so that only some of the sub-totals will be included. This partial rollup uses the following syntax:

```
GROUP BY expr1, ROLLUP(expr2, expr3);
```

In this case, the `GROUP BY` clause creates subtotals at (2+1=3) aggregation levels. That is, at level (`expr1`, `expr2`, `expr3`), (`expr1`, `expr2`), and (`expr1`).

Example 21–3 Partial ROLLUP

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
   TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY channel_desc, ROLLUP(calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
```

```
Internet             2000-09  GB          16,569
Internet             2000-09  US         124,224
Internet             2000-09             140,793
Internet             2000-10  GB          14,539
Internet             2000-10  US         137,054
Internet             2000-10             151,593
Internet                                 292,387
Direct Sales         2000-09  GB          85,223
Direct Sales         2000-09  US         638,201
Direct Sales         2000-09             723,424
Direct Sales         2000-10  GB          91,925
Direct Sales         2000-10  US         682,297
Direct Sales         2000-10             774,222
Direct Sales                           1,497,646
```

This query returns the following sets of rows:

- Regular aggregation rows that would be produced by `GROUP BY` without using `ROLLUP`.
- First-level subtotals aggregating across `country_id` for each combination of `channel_desc` and `calendar_month_desc`.
- Second-level subtotals aggregating across `calendar_month_desc` and `country_id` for each `channel_desc` value.
- It does not produce a grand total row.

CUBE Extension to GROUP BY

`CUBE` takes a specified set of grouping columns and creates subtotals for all of their possible combinations. In terms of multidimensional analysis, `CUBE` generates all the subtotals that could be calculated for a data cube with the specified dimensions. If you have specified `CUBE(time, region, department)`, the result set will include all the values that would be included in an equivalent `ROLLUP` statement plus additional combinations. For instance, in Figure 21–1, the departmental totals across regions (279,000 and 319,000) would not be calculated by a `ROLLUP(time, region, department)` clause, but they would be calculated by a `CUBE(time, region, department)` clause. If *n* columns are specified for a `CUBE`, there will be 2 to the *n* combinations of subtotals returned. Example 21–4 on page 21-8 gives an example of a three-dimension cube.

> **See Also:** *Oracle Database SQL Language Reference* for syntax and restrictions

When to Use CUBE

Consider Using `CUBE` in any situation requiring cross-tabular reports. The data needed for cross-tabular reports can be generated with a single `SELECT` using `CUBE`. Like `ROLLUP`, `CUBE` can be helpful in generating summary tables. Note that population of summary tables is even faster if the `CUBE` query executes in parallel.

`CUBE` is typically most suitable in queries that use columns from multiple dimensions rather than columns representing different levels of a single dimension. For instance, a commonly requested cross-tabulation might need subtotals for all the combinations of month, state, and product. These are three independent dimensions, and analysis of all possible subtotal combinations is commonplace. In contrast, a cross-tabulation showing all possible combinations of year, month, and day would have several values of limited interest, because there is a natural hierarchy in the time dimension.

Subtotals such as profit by day of month summed across year would be unnecessary in most analyses. Relatively few users need to ask "What were the total sales for the 16th of each month across the year?" See "Hierarchy Handling in ROLLUP and CUBE" on page 21-20 for an example of handling rollup calculations efficiently.

CUBE Syntax

CUBE appears in the GROUP BY clause in a SELECT statement. Its form is:

```
SELECT …  GROUP BY CUBE (grouping_column_reference_list)
```

Example 21–4 CUBE

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
      TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id
 AND customers.country_id = countries.country_id
 AND channels.channel_desc IN
  ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
  ('2000-09', '2000-10') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
                                    1,790,032
                              GB      208,257
                              US    1,581,775
                     2000-09          864,217
                     2000-09  GB      101,792
                     2000-09  US      762,425
                     2000-10          925,815
                     2000-10  GB      106,465
                     2000-10  US      819,351
Internet                              292,387
Internet                      GB       31,109
Internet                      US      261,278
Internet             2000-09          140,793
Internet             2000-09  GB       16,569
Internet             2000-09  US      124,224
Internet             2000-10          151,593
Internet             2000-10  GB       14,539
Internet             2000-10  US      137,054
Direct Sales                        1,497,646
Direct Sales                  GB      177,148
Direct Sales                  US    1,320,497
Direct Sales         2000-09          723,424
Direct Sales         2000-09  GB       85,223
Direct Sales         2000-09  US      638,201
Direct Sales         2000-10          774,222
Direct Sales         2000-10  GB       91,925
Direct Sales         2000-10  US      682,297
```

This query illustrates CUBE aggregation across three dimensions.

Partial CUBE

Partial CUBE resembles partial ROLLUP in that you can limit it to certain dimensions and precede it with columns outside the CUBE operator. In this case, subtotals of all

possible combinations are limited to the dimensions within the cube list (in parentheses), and they are combined with the preceding items in the GROUP BY list.

The syntax for partial CUBE is as follows:

```
GROUP BY expr1, CUBE(expr2, expr3)
```

This syntax example calculates 2*2, or 4, subtotals. That is:

- (expr1, expr2, expr3)
- (expr1, expr2)
- (expr1, expr3)
- (expr1)

Example 21–5 Partial CUBE

Using the sales database, you can issue the following statement:

```
SELECT channel_desc, calendar_month_desc, countries.country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id = times.time_id
  AND sales.cust_id = customers.cust_id
  AND customers.country_id=countries.country_id
  AND sales.channel_id = channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY channel_desc, CUBE(calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet                                292,387
Internet                        GB       31,109
Internet                        US      261,278
Internet             2000-09            140,793
Internet             2000-09    GB       16,569
Internet             2000-09    US      124,224
Internet             2000-10            151,593
Internet             2000-10    GB       14,539
Internet             2000-10    US      137,054
Direct Sales                          1,497,646
Direct Sales                    GB      177,148
Direct Sales                    US    1,320,497
Direct Sales         2000-09            723,424
Direct Sales         2000-09    GB       85,223
Direct Sales         2000-09    US      638,201
Direct Sales         2000-10            774,222
Direct Sales         2000-10    GB       91,925
Direct Sales         2000-10    US      682,297
```

Calculating Subtotals Without CUBE

Just as for ROLLUP, multiple SELECT statements combined with UNION ALL statements could provide the same information gathered through CUBE. However, this might require many SELECT statements. For an n-dimensional cube, 2 to the *n* SELECT statements are needed. In the three-dimension example, this would mean issuing SELECT statements linked with UNION ALL. So many SELECT statements yield inefficient processing and very lengthy SQL.

Consider the impact of adding just one more dimension when calculating all possible combinations: the number of SELECT statements would double to 16. The more columns used in a CUBE clause, the greater the savings compared to the UNION ALL approach.

GROUPING Functions

Two challenges arise with the use of ROLLUP and CUBE. First, how can you programmatically determine which result set rows are subtotals, and how do you find the exact level of aggregation for a given subtotal? You often need to use subtotals in calculations such as percent-of-totals, so you need an easy way to determine which rows are the subtotals. Second, what happens if query results contain both stored NULL values and "NULL" values created by a ROLLUP or CUBE? How can you differentiate between the two? This section discusses some of these situations.

> **See Also:** *Oracle Database SQL Language Reference* for syntax and restrictions

GROUPING Function

GROUPING handles these problems. Using a single column as its argument, GROUPING returns 1 when it encounters a NULL value created by a ROLLUP or CUBE operation. That is, if the NULL indicates the row is a subtotal, GROUPING returns a 1. Any other type of value, including a stored NULL, returns a 0.

GROUPING appears in the selection list portion of a SELECT statement. Its form is:

```
SELECT …  [GROUPING(dimension_column)…]  …
  GROUP BY …    {CUBE | ROLLUP| GROUPING SETS}  (dimension_column)
```

Example 21–6 GROUPING to Mask Columns

This example uses GROUPING to create a set of mask columns for the result set shown in Example 21–3 on page 21-6. The mask columns are easy to analyze programmatically.

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) AS Ch,
   GROUPING(calendar_month_desc) AS Mo, GROUPING(country_iso_code) AS Co
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id
  AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND countries.country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, countries.country_iso_code);

CHANNEL_DESC         CALENDAR CO SALES$                CH         MO         CO
-------------------- -------- -- -------------- ---------- ---------- ----------
Internet             2000-09  GB          16,569          0          0          0
Internet             2000-09  US         124,224          0          0          0
Internet             2000-09             140,793          0          0          1
Internet             2000-10  GB          14,539          0          0          0
Internet             2000-10  US         137,054          0          0          0
Internet             2000-10             151,593          0          0          1
Internet                                 292,387          0          1          1
Direct Sales         2000-09  GB          85,223          0          0          0
```

```
Direct Sales         2000-09  US        638,201          0          0          0
Direct Sales         2000-09            723,424          0          0          1
Direct Sales         2000-10  GB         91,925          0          0          0
Direct Sales         2000-10  US        682,297          0          0          0
Direct Sales         2000-10            774,222          0          0          1
Direct Sales                          1,497,646          0          1          1
                                      1,790,032          1          1          1
```

A program can easily identify the detail rows by a mask of "0 0 0" on the T, R, and D columns. The first level subtotal rows have a mask of "0 0 1", the second level subtotal rows have a mask of "0 1 1", and the overall total row has a mask of "1 1 1".

You can improve the readability of result sets by using the GROUPING and DECODE functions as shown in Example 21–7.

Example 21–7 GROUPING For Readability

```
SELECT DECODE(GROUPING(channel_desc), 1, 'Multi-channel sum', channel_desc) AS
 Channel, DECODE (GROUPING (country_iso_code), 1, 'Multi-country sum',
 country_iso_code) AS Country, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id
  AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc= '2000-09'
  AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, country_iso_code);

CHANNEL              COUNTRY           SALES$
-------------------- ----------------- --------------
Multi-channel sum    Multi-country sum        864,217
Multi-channel sum    GB                       101,792
Multi-channel sum    US                       762,425
Internet             Multi-country sum        140,793
Internet             GB                        16,569
Internet             US                       124,224
Direct Sales         Multi-country sum        723,424
Direct Sales         GB                        85,223
Direct Sales         US                       638,201
```

To understand the previous statement, note its first column specification, which handles the channel_desc column. Consider the first line of the previous statement:

```
SELECT DECODE(GROUPING(channel_desc), 1, 'All Channels', channel_desc)AS Channel
```

In this, the channel_desc value is determined with a DECODE function that contains a GROUPING function. The GROUPING function returns a 1 if a row value is an aggregate created by ROLLUP or CUBE, otherwise it returns a 0. The DECODE function then operates on the GROUPING function's results. It returns the text "All Channels" if it receives a 1 and the channel_desc value from the database if it receives a 0. Values from the database will be either a real value such as "Internet" or a stored NULL. The second column specification, displaying country_id, works the same way.

When to Use GROUPING

The GROUPING function is not only useful for identifying NULLs, it also enables sorting subtotal rows and filtering results. In Example 21–8, you retrieve a subset of

the subtotals created by a CUBE and none of the base-level aggregations. The HAVING clause constrains columns that use GROUPING functions.

Example 21–8 GROUPING Combined with HAVING

```
SELECT channel_desc, calendar_month_desc, country_iso_code, TO_CHAR(
SUM(amount_sold), '9,999,999,999') SALES$, GROUPING(channel_desc) CH, GROUPING
  (calendar_month_desc)  MO, GROUPING(country_iso_code) CO
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING (GROUPING(channel_desc)=1 AND GROUPING(calendar_month_desc)= 1
  AND GROUPING(country_iso_code)=1) OR (GROUPING(channel_desc)=1
  AND GROUPING (calendar_month_desc)= 1) OR (GROUPING(country_iso_code)=1
  AND GROUPING(calendar_month_desc)= 1);

CHANNEL_DESC         C CO SALES$                CH         MO         CO
-------------------- - -- -------------- ---------- ---------- ----------
                       US      1,581,775          1          1          0
                       GB        208,257          1          1          0
Direct Sales                   1,497,646          0          1          1
Internet                         292,387          0          1          1
                               1,790,032          1          1          1
```

Compare the result set of Example 21–8 with that in Example 21–3 on page 21-6 to see how Example 21–8 is a precisely specified group: it contains only the yearly totals, regional totals aggregated over time and department, and the grand total.

GROUPING_ID Function

To find the GROUP BY level of a particular row, a query must return GROUPING function information for each of the GROUP BY columns. If we do this using the GROUPING function, every GROUP BY column requires another column using the GROUPING function. For instance, a four-column GROUP BY clause must be analyzed with four GROUPING functions. This is inconvenient to write in SQL and increases the number of columns required in the query. When you want to store the query result sets in tables, as with materialized views, the extra columns waste storage space.

To address these problems, you can use the GROUPING_ID function. GROUPING_ID returns a single number that enables you to determine the exact GROUP BY level. For each row, GROUPING_ID takes the set of 1's and 0's that would be generated if you used the appropriate GROUPING functions and concatenates them, forming a bit vector. The bit vector is treated as a binary number, and the number's base-10 value is returned by the GROUPING_ID function. For instance, if you group with the expression CUBE(a, b) the possible values are as shown in Table 21–2.

Table 21–2 GROUPING_ID Example for CUBE(a, b)

Aggregation Level	Bit Vector	GROUPING_ID
a, b	0 0	0
a	0 1	1
b	1 0	2

Table 21–2 (Cont.) GROUPING_ID Example for CUBE(a, b)

Aggregation Level	Bit Vector	GROUPING_ID
Grand Total	1 1	3

GROUPING_ID clearly distinguishes groupings created by grouping set specification, and it is very useful during refresh and rewrite of materialized views.

GROUP_ID Function

While the extensions to GROUP BY offer power and flexibility, they also allow complex result sets that can include duplicate groupings. The GROUP_ID function lets you distinguish among duplicate groupings. If there are multiple sets of rows calculated for a given level, GROUP_ID assigns the value of 0 to all the rows in the first set. All other sets of duplicate rows for a particular grouping are assigned higher values, starting with 1. For example, consider the following query, which generates a duplicate grouping:

Example 21–9 GROUP_ID

```
SELECT country_iso_code, SUBSTR(cust_state_province,1,12), SUM(amount_sold),
  GROUPING_ID(country_iso_code, cust_state_province) GROUPING_ID, GROUP_ID()
FROM sales, customers, times, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id=countries.country_id AND times.time_id= '30-OCT-00'
  AND country_iso_code IN ('FR', 'ES')
GROUP BY GROUPING SETS (country_iso_code,
ROLLUP(country_iso_code, cust_state_province));

CO SUBSTR(CUST_ SUM(AMOUNT_SOLD) GROUPING_ID GROUP_ID()
-- ------------ ---------------- ----------- ----------
ES Alicante               135.32           0          0
ES Valencia              4133.56           0          0
ES Barcelona               24.22           0          0
FR Centre                   74.3           0          0
FR Aquitaine              231.97           0          0
FR Rhtne-Alpes           1624.69           0          0
FR Ile-de-Franc          1860.59           0          0
FR Languedoc-Ro           4287.4           0          0
                         12372.05           3          0
ES                        4293.1           1          0
FR                       8078.95           1          0
ES                        4293.1           1          1
FR                       8078.95           1          1
```

This query generates the following groupings: (country_id, cust_state_province), (country_id), (country_id), and (). Note that the grouping (country_id) is repeated twice. The syntax for GROUPING SETS is explained in "GROUPING SETS Expression" on page 21-14.

This function helps you filter out duplicate groupings from the result. For example, you can filter out duplicate (region) groupings from the previous example by adding a HAVING clause condition GROUP_ID()=0 to the query.

GROUPING SETS Expression

You can selectively specify the set of groups that you want to create using a GROUPING SETS expression within a GROUP BY clause. This allows precise specification across multiple dimensions without computing the whole CUBE. For example, you can say:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY GROUPING SETS((channel_desc, calendar_month_desc, country_iso_code),
    (channel_desc, country_iso_code), (calendar_month_desc, country_iso_code));
```

Note that this statement uses composite columns, described in "Composite Columns" on page 21-15. This statement calculates aggregates over three groupings:

- (channel_desc, calendar_month_desc, country_iso_code)
- (channel_desc, country_iso_code)
- (calendar_month_desc, country_iso_code)

Compare the previous statement with the following alternative, which uses the CUBE operation and the GROUPING_ID function to return the desired rows:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
       GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code) gid
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY CUBE(channel_desc, calendar_month_desc, country_iso_code)
HAVING GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=0
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=2
  OR GROUPING_ID(channel_desc, calendar_month_desc, country_iso_code)=4;
```

This statement computes all the 8 (2 *2 *2) groupings, though only the previous 3 groups are of interest to you.

Another alternative is the following statement, which is lengthy due to several unions. This statement requires three scans of the base table, making it inefficient. CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. For example, consider the following statement:

```
CUBE(a, b, c)
```

This statement is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
ROLLUP(a, b, c)
```

And this statement is equivalent to:

```
GROUPING SETS ((a, b, c), (a, b), ())
```

GROUPING SETS Syntax

GROUPING SETS syntax lets you define multiple groupings in the same query. GROUP BY computes all the groupings specified and combines them with UNION ALL. For example, consider the following statement:

```
GROUP BY GROUPING sets (channel_desc, calendar_month_desc, country_id )
```

This statement is equivalent to:

```
GROUP BY channel_desc UNION ALL
GROUP BY calendar_month_desc UNION ALL GROUP BY country_id
```

Table 21–3 shows grouping sets specification and equivalent GROUP BY specification. Note that some examples use composite columns.

Table 21–3 GROUPING SETS Statements and Equivalent GROUP BY

GROUPING SETS Statement	Equivalent GROUP BY Statement
`GROUP BY GROUPING SETS(a, b, c)`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c`
`GROUP BY GROUPING SETS(a, b, (b, c))`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c`
`GROUP BY GROUPING SETS((a, b, c))`	`GROUP BY a, b, c`
`GROUP BY GROUPING SETS(a, (b), ())`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()`
`GROUP BY GROUPING SETS(a, ROLLUP(b, c))`	`GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)`

In the absence of an optimizer that looks across query blocks to generate the execution plan, a query based on UNION would need multiple scans of the base table, sales. This could be very inefficient as fact tables will normally be huge. Using GROUPING SETS statements, all the groupings of interest are available in the same query block.

Composite Columns

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement:

```
ROLLUP (year, (quarter, month), day)
```

In this statement, the data is not rolled up across year and quarter, but is instead equivalent to the following groupings of a UNION ALL:

- (year, quarter, month, day),
- (year, quarter, month),
- (year)
- ()

Here, (quarter, month) form a composite column and are treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, GROUPING SETS, and

concatenated groupings. For example, in CUBE or ROLLUP, composite columns would mean skipping aggregation across certain levels. That is, the following statement:

```
GROUP BY ROLLUP(a, (b, c))
```

This is equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Here, (b, c) are treated as a unit and rollup will not be applied across (b, c). It is as if you have an alias, for example z, for (b, c) and the GROUP BY expression reduces to GROUP BY ROLLUP(a, z). Compare this with the normal rollup as in the following:

```
GROUP BY ROLLUP(a, b, c)
```

This would be the following:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ().
```

Similarly, the following statement is equivalent to the four GROUP BYs:

```
GROUP BY CUBE((a, b), c)

GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP By ()
```

In GROUPING SETS, a composite column is used to denote a particular level of GROUP BY. See Table 21–3 for more examples of composite columns.

Example 21–10 Composite Columns

You do not have full control over what aggregation levels you want with CUBE and ROLLUP. For example, consider the following statement:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
 TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id
  AND sales.channel_id= channels.channel_id
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, calendar_month_desc, country_iso_code);
```

This statement results in Oracle computing the following groupings:

- (channel_desc, calendar_month_desc, country_iso_code)
- (channel_desc, calendar_month_desc)
- (channel_desc)
- ()

If you are just interested in the first, third, and fourth of these groupings, you cannot limit the calculation to those groupings without using composite columns. With

composite columns, this is possible by treating month and country as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing CUBE and ROLLUP. Thus, you would say:

```
SELECT channel_desc, calendar_month_desc, country_iso_code,
    TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
      sales.channel_id= channels.channel_id  AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(channel_desc, (calendar_month_desc, country_iso_code));

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Internet             2000-09  GB         228,241
Internet             2000-09  US         228,241
Internet             2000-10  GB         239,236
Internet             2000-10  US         239,236
Internet                                 934,955
Direct Sales         2000-09  GB       1,217,808
Direct Sales         2000-09  US       1,217,808
Direct Sales         2000-10  GB       1,225,584
Direct Sales         2000-10  US       1,225,584
Direct Sales                             4,886,784
                                         5,821,739
```

Concatenated Groupings

Concatenated groupings offer a concise way to generate useful combinations of groupings. Groupings specified with concatenated groupings yield the cross-product of groupings from each grouping set. The cross-product operation enables even a small number of concatenated groupings to generate a large number of final groups. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, and rollups, and separating them with commas. Here is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL defines the following groupings:

```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- Ease of query development

  You need not enumerate all groupings manually.

- Use by applications

  SQL generated by analytical applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension.

Example 21–11 Concatenated Groupings

You can also specify more than one grouping in the GROUP BY clause. For example, if you want aggregated sales values for each product rolled up across all levels in the time dimension (year, month and day), and across all levels in the geography dimension (region), you can issue the following statement:

```
SELECT channel_desc, calendar_year, calendar_quarter_desc, country_iso_code,
```

```
  cust_state_province, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id
 AND sales.channel_id = channels.channel_id AND countries.country_id =
    customers.country_id AND channels.channel_desc IN
   ('Direct Sales', 'Internet') AND times.calendar_month_desc IN ('2000-09',
 '2000-10') AND countries.country_iso_code IN ('GB', 'FR')
GROUP BY channel_desc, GROUPING SETS (ROLLUP(calendar_year,
   calendar_quarter_desc),
ROLLUP(country_iso_code, cust_state_province));
```

This results in the following groupings:

- (channel_desc, calendar_year, calendar_quarter_desc)
- (channel_desc, calendar_year)
- (channel_desc)
- (channel_desc, country_iso_code, cust_state_province)
- (channel_desc, country_iso_code)
- (channel_desc)

This is the cross-product of the following:

- The expression, channel_desc
- ROLLUP(calendar_year, calendar_quarter_desc), which is equivalent to ((calendar_year, calendar_quarter_desc), (calendar_year), ())
- ROLLUP(country_iso_code, cust_state_province), which is equivalent to ((country_iso_code, cust_state_province), (country_iso_code), ())

Note that the output contains two occurrences of (channel_desc) group. To filter out the extra (channel_desc) group, the query could use a GROUP_ID function.

Another concatenated join example is Example 21–12, showing the cross product of two grouping sets.

Example 21–12 Concatenated Groupings (Cross-Product of Two Grouping Sets)

```
SELECT country_iso_code, cust_state_province, calendar_year,
calendar_quarter_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
 countries.country_id=customers.country_id  AND
  sales.channel_id= channels.channel_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND country_iso_code IN ('GB', 'FR')
GROUP BY GROUPING SETS (country_iso_code, cust_state_province),
         GROUPING SETS (calendar_year, calendar_quarter_desc);
```

This statement results in the computation of groupings:

- (country_iso_code, year), (country_iso_code, calendar_quarter_desc), (cust_state_province, year) and (cust_state_province, calendar_quarter_desc)

Concatenated Groupings and Hierarchical Data Cubes

One of the most important uses for concatenated groupings is to generate the aggregates needed for a hierarchical cube of data. A hierarchical cube is a data set

where the data is aggregated along the rollup hierarchy of each of its dimensions and these aggregations are combined across dimensions. It includes the typical set of aggregations needed for business intelligence queries. By using concatenated groupings, you can generate all the aggregations needed by a hierarchical cube with just *n* ROLLUPs (where *n* is the number of dimensions), and avoid generating unwanted aggregations.

Consider just three of the dimensions in the sh sample schema data set, each of which has a multilevel hierarchy:

- time: year, quarter, month, day (week is in a separate hierarchy)
- product: category, subcategory, prod_name
- geography: region, subregion, country, state, city

This data is represented using a column for each level of the hierarchies, creating a total of twelve columns for dimensions, plus the columns holding sales figures.

For our business intelligence needs, we would like to calculate and store certain aggregates of the various combinations of dimensions. In Example 21–13 on page 21-19, we create the aggregates for all levels, except for "day", which would create too many rows. In particular, we want to use ROLLUP within each dimension to generate useful aggregates. Once we have the ROLLUP-based aggregates within each dimension, we want to combine them with the other dimensions. This will generate our hierarchical cube. Note that this is not at all the same as a CUBE using all twelve of the dimension columns: that would create 2 to the 12th power (4,096) aggregation groups, of which we need only a small fraction. Concatenated grouping sets make it easy to generate exactly the aggregations we need. Example 21–13 shows where a GROUP BY clause is needed.

Example 21–13 Concatenated Groupings and Hierarchical Cubes

```
SELECT calendar_year, calendar_quarter_desc, calendar_month_desc,
  country_region, country_subregion, countries.country_iso_code,
 cust_state_province, cust_city, prod_category_desc, prod_subcategory_desc,
 prod_name, TO_CHAR(SUM
 (amount_sold), '9,999,999,999') SALES$
FROM sales, customers, times, channels, countries, products
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id AND
  sales.channel_id= channels.channel_id AND sales.prod_id=products.prod_id AND
  customers.country_id=countries.country_id AND channels.channel_desc IN
 ('Direct Sales', 'Internet') AND times.calendar_month_desc IN
 ('2000-09', '2000-10') AND prod_name IN ('Envoy Ambassador',
 'Mouse Pad') AND countries.country_iso_code IN ('GB', 'US')
GROUP BY ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc),
  ROLLUP(country_region, country_subregion, countries.country_iso_code,
         cust_state_province, cust_city),
  ROLLUP(prod_category_desc, prod_subcategory_desc, prod_name);
```

The rollups in the GROUP BY specification generate the following groups, four for each dimension.

Table 21–4 Hierarchical CUBE Example

ROLLUP By Time	ROLLUP By Product	ROLLUP By Geography
year, quarter, month	category, subcategory, name	region, subregion, country, state, city
		region, subregion, country, state
		region, subregion, country
year, quarter	category, subcategory	region, subregion
year	category	region
all times	all products	all geographies

The concatenated grouping sets specified in the previous SQL will take the ROLLUP aggregations listed in the table and perform a cross-product on them. The cross-product will create the 96 (4x4x6) aggregate groups needed for a hierarchical cube of the data. There are major advantages in using three ROLLUP expressions to replace what would otherwise require 96 grouping set expressions: the concise SQL is far less error-prone to develop and far easier to maintain, and it enables much better query optimization. You can picture how a cube with more dimensions and more levels would make the use of concatenated groupings even more advantageous.

See "Working with Hierarchical Cubes in SQL" on page 21-22 for more information regarding hierarchical cubes.

Considerations when Using Aggregation

This section discusses the following topics.

Hierarchy Handling in ROLLUP and CUBE

The ROLLUP and CUBE extensions work independently of any hierarchy metadata in your system. Their calculations are based entirely on the columns specified in the SELECT statement in which they appear. This approach enables CUBE and ROLLUP to be used whether or not hierarchy metadata is available. The simplest way to handle levels in hierarchical dimensions is by using the ROLLUP extension and indicating levels explicitly through separate columns. The following code shows a simple example of this with months rolled up to quarters and quarters rolled up to years.

Example 21–14 ROLLUP and CUBE Hierarchy Handling

```
SELECT calendar_year, calendar_quarter_number,
    calendar_month_number, SUM(amount_sold)
FROM sales, times, products, customers, countries
WHERE sales.time_id=times.time_id
  AND sales.prod_id=products.prod_id
  AND customers.country_id = countries.country_id
  AND sales.cust_id=customers.cust_id
  AND prod_name IN ('Envoy Ambassador', 'Mouse Pad')
  AND country_iso_code = 'GB' AND calendar_year=1999
GROUP BY ROLLUP(calendar_year, calendar_quarter_number, calendar_month_number);
```

```
CALENDAR_YEAR CALENDAR_QUARTER_NUMBER CALENDAR_MONTH_NUMBER SUM(AMOUNT_SOLD)
------------- ----------------------- --------------------- ----------------
         1999                       1                     1          5521.34
         1999                       1                     2         22232.95
         1999                       1                     3         10672.63
         1999                       1                               38426.92
         1999                       2                     4         23658.05
         1999                       2                     5          5766.31
         1999                       2                     6         23939.32
         1999                       2                               53363.68
         1999                       3                     7         12132.18
         1999                       3                     8         13128.96
         1999                       3                     9         19571.96
         1999                       3                                44833.1
         1999                       4                    10         15752.18
         1999                       4                    11          7011.21
         1999                       4                    12          14257.5
         1999                       4                               37020.89
         1999                                                      173644.59
                                                                   173644.59
```

Column Capacity in ROLLUP and CUBE

CUBE, ROLLUP, and GROUPING SETS do not restrict the GROUP BY clause column capacity. The GROUP BY clause, with or without the extensions, can work with up to 255 columns. However, the combinatorial explosion of CUBE makes it unwise to specify a large number of columns with the CUBE extension. Consider that a 20-column list for CUBE would create 2 to the 20 combinations in the result set. A very large CUBE list could strain system resources, so any such query must be tested carefully for performance and the load it places on the system.

HAVING Clause Used with GROUP BY Extensions

The HAVING clause of SELECT statements is unaffected by the use of GROUP BY. Note that the conditions specified in the HAVING clause apply to both the subtotal and non-subtotal rows of the result set. In some cases a query may need to exclude the subtotal rows or the non-subtotal rows from the HAVING clause. This can be achieved by using a GROUPING or GROUPING_ID function together with the HAVING clause. See Example 21–8 on page 21-12 and its associated SQL statement for an example.

ORDER BY Clause Used with GROUP BY Extensions

In many cases, a query must order the rows in a certain way, and this is done with the ORDER BY clause. The ORDER BY clause of a SELECT statement is unaffected by the use of GROUP BY, since the ORDER BY clause is applied after the GROUP BY calculations are complete.

Note that the ORDER BY specification makes no distinction between aggregate and non-aggregate rows of the result set. For instance, you might wish to list sales figures in declining order, but still have the subtotals at the end of each group. Simply ordering sales figures in descending sequence will not be sufficient, since that will place the subtotals (the largest values) at the start of each group. Therefore, it is essential that the columns in the ORDER BY clause include columns that differentiate aggregate from non-aggregate columns. This requirement means that queries using ORDER BY along with aggregation extensions to GROUP BY will generally need to use one or more of the GROUPING functions.

Using Other Aggregate Functions with ROLLUP and CUBE

The examples in this chapter show ROLLUP and CUBE used with the SUM function. While this is the most common type of aggregation, these extensions can also be used with all other functions available to the GROUP BY clause, for example, COUNT, AVG, MIN, MAX, STDDEV, and VARIANCE. COUNT, which is often needed in cross-tabular analyses, is likely to be the second most commonly used function.

Computation Using the WITH Clause

The WITH clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query. WITH is a part of the SQL-99 standard. This is particularly useful when a query has multiple references to the same query block and there are joins and aggregations. Using the WITH clause, Oracle retrieves the results of a query block and stores them in the user's temporary tablespace. Note that Oracle Database does not support recursive use of the WITH clause. Note that Oracle Database supports recursive use of the WITH clause that may be used for such queries as are used with a bill of materials or expansion of parent-child hierarchies to parent-descendant hierarchies. See *Oracle Database SQL Language Reference* for more information.

The following query is an example of where you can improve performance and write SQL more simply by using the WITH clause. The query calculates the sum of sales for each channel and holds it under the name channel_summary. Then it checks each channel's sales total to see if any channel's sales are greater than one third of the total sales. By using the WITH clause, the channel_summary data is calculated just once, avoiding an extra scan through the large sales table.

Example 21–15 WITH Clause

```
WITH channel_summary AS (SELECT channels.channel_desc, SUM(amount_sold)
AS channel_total FROM sales, channels
WHERE sales.channel_id = channels.channel_id GROUP BY channels.channel_desc)
SELECT channel_desc, channel_total
FROM channel_summary WHERE channel_total > (SELECT SUM(channel_total) * 1/3
FROM channel_summary);

CHANNEL_DESC         CHANNEL_TOTAL
-------------------- -------------
Direct Sales            57875260.6
```

Note that this example could also be performed efficiently using the reporting aggregate functions described in Chapter 22, "SQL for Analysis and Reporting".

Working with Hierarchical Cubes in SQL

This section illustrates examples of working with hierarchical cubes.

Specifying Hierarchical Cubes in SQL

Oracle Database can specify hierarchical cubes in a simple and efficient SQL query. These hierarchical cubes represent the logical cubes referred to in many analytical SQL products. To specify data in the form of hierarchical cubes, you can use one of the extensions to the GROUP BY clause, concatenated grouping sets, to generate the aggregates needed for a hierarchical cube of data. By using concatenated rollup (rolling up along the hierarchy of each dimension and then concatenate them across

multiple dimensions), you can generate all the aggregations needed by a hierarchical cube.

Example 21–16 Concatenated ROLLUP

The following shows the GROUP BY clause needed to create a hierarchical cube for a 2-dimensional example similar to Example 21–13. The following simple syntax performs a concatenated rollup:

```
GROUP BY ROLLUP(year, quarter, month), ROLLUP(Division, brand, item)
```

This concatenated rollup takes the ROLLUP aggregations similar to those listed in Table 21–4, " Hierarchical CUBE Example" in the prior section and performs a cross-product on them. The cross-product will create the 16 (4x4) aggregate groups needed for a hierarchical cube of the data.

Querying Hierarchical Cubes in SQL

Analytic applications treat data as cubes, but they want only certain slices and regions of the cube. Concatenated rollup (hierarchical cube) enables relational data to be treated as cubes. To handle complex analytic queries, the fundamental technique is to enclose a hierarchical cube query in an outer query that specifies the exact slice needed from the cube. Oracle Database optimizes the processing of hierarchical cubes nested inside slicing queries. By applying many powerful algorithms, these queries can be processed at unprecedented speed and scale. This enables SQL analytical tools and applications to use a consistent style of queries to handle the most complex questions.

Example 21–17 Hierarchical Cube Query

Consider the following analytic query. It consists of a hierarchical cube query nested in a slicing query.

```
SELECT month, division, sum_sales FROM
  (SELECT year, quarter, month, division, brand, item, SUM(sales) sum_sales,
     GROUPING_ID(grouping-columns) gid
   FROM sales, products, time
   WHERE join-condition
   GROUP BY ROLLUP(year, quarter, month),
            ROLLUP(division, brand, item))
WHERE division = 25 AND month = 200201 AND gid = gid-for-Division-Month;
```

The inner hierarchical cube specified defines a simple cube, with two dimensions and four levels in each dimension. It would generate 16 groups (4 Time levels * 4 Product levels). The GROUPING_ID function in the query identifies the specific group each row belongs to, based on the aggregation level of the *grouping-columns* in its argument.

The outer query applies the constraints needed for our specific query, limiting Division to a value of 25 and Month to a value of 200201 (representing January 2002 in this case). In conceptual terms, it slices a small chunk of data from the cube. The outer query's constraint on the GID column, indicated in the query by *gid-for-division-month* would be the value of a key indicating that the data is grouped as a combination of division and month. The GID constraint selects only those rows that are aggregated at the level of a GROUP BY month, division clause.

Oracle Database removes unneeded aggregation groups from query processing based on the outer query conditions. The outer conditions of the previous query limit the result set to a single group aggregating division and month. Any other groups involving year, month, brand, and item are unnecessary here. The group pruning optimization recognizes this and transforms the query into:

```
SELECT month, division, sum_sales
FROM (SELECT  null, null,  month, division, null, null, SUM(sales) sum_sales,
      GROUPING_ID(grouping-columns) gid
      FROM sales, products, time WHERE join-condition
   GROUP BY month, division)
WHERE division = 25 AND month = 200201 AND gid = gid-for-Division-Month;
```

The bold items highlight the changed SQL. The inner query now has a simple `GROUP BY` clause of `month`, `division`. The columns `year`, `quarter`, `brand`, and `item` have been converted to null to match the simplified `GROUP BY` clause. Because the query now requests just one group, fifteen out of sixteen groups are removed from the processing, greatly reducing the work. For a cube with more dimensions and more levels, the savings possible through group pruning can be far greater. Note that the group pruning transformation works with all the `GROUP BY` extensions: `ROLLUP`, `CUBE`, and `GROUPING SETS`.

While the optimizer has simplified the previous query to a simple `GROUP BY`, faster response times can be achieved if the group is precomputed and stored in a materialized view. Because online analytical queries can ask for any slice of the cube many groups may need to be precomputed and stored in a materialized view. This is discussed in the next section.

SQL for Creating Materialized Views to Store Hierarchical Cubes

Analytical SQL requires fast response times for multiple users, and this in turn demands that significant parts of a cube be precomputed and held in materialized views.

Data warehouse designers can choose exactly how much data to materialize. A data warehouse can have the full hierarchical cube materialized. While this will take the most storage space, it ensures quick response for any query within the cube. Alternatively, a data warehouse could have just partial materialization, saving storage space, but allowing only a subset of possible queries to be answered at highest speed. If the queries cover the full range of aggregate groupings possible in its data set, it may be best to materialize the whole hierarchical cube.

This means that each dimension's aggregation hierarchy is precomputed in combination with each of the other dimensions. Naturally, precomputing a full hierarchical cube requires more disk space and higher creation and refresh times than a small set of aggregate groups. The trade-off in processing time and disk space versus query performance must be considered before deciding to create it. An additional possibility you could consider is to use data compression to lessen your disk space requirements.

See Also:

- *Oracle Database SQL Language Reference* for table compression syntax and restrictions
- *Oracle Database Administrator's Guide* for further details about table compression
- "Storage And Table Compression" on page 9-16 for details regarding table compression

Examples of Hierarchical Cube Materialized Views

This section shows complete and partial hierarchical cube materialized views. Many of the examples are meant to illustrate capabilities, and do not actually run.

In a data warehouse where rolling window scenario is very common, it is recommended that you store the hierarchical cube in multiple materialized views - one for each level of time you are interested in. Hence, a complete hierarchical cube will be stored in four materialized views: `sales_hierarchical_mon_cube_mv`, `sales_hierarchical_qtr_cube_mv`, `sales_hierarchical_yr_cube_mv`, and `sales_hierarchical_all_cube_mv`.

The following statements create a complete hierarchical cube stored in a set of three composite partitioned and one list partitioned materialized view.

Example 21–18 Complete Hierarchical Cube Materialized View

```
CREATE MATERIALIZED VIEW sales_hierarchical_mon_cube_mv
PARTITION BY RANGE (mon)
SUBPARTITION BY LIST (gid)
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, calendar_quarter_desc qtr, calendar_month_desc mon,
    country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, calendar_quarter_desc, calendar_month_desc,
                country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales,
    COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, calendar_quarter_desc, calendar_month_desc,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
...;

CREATE MATERIALIZED VIEW sales_hierarchical_qtr_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, calendar_quarter_desc qtr,
    country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(calendar_year, calendar_quarter_desc,
                country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales,
    COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
      AND s.time_id = t.time_id
GROUP BY calendar_year, calendar_quarter_desc,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
PARTITION BY RANGE (qtr)
 SUBPARTITION BY LIST (gid)
...;

CREATE MATERIALIZED VIEW sales_hierarchical_yr_cube_mv
PARTITION BY RANGE (year)
SUBPARTITION BY LIST (gid)
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
```

```
    GROUPING_ID(calendar_year, country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year,
  ROLLUP(country_id, cust_state_province, cust_city),
  ROLLUP(prod_category, prod_subcategory, prod_name),
...;

CREATE MATERIALIZED VIEW sales_hierarchical_all_cube_mv
REFRESH FAST ON DEMAND
ENABLE QUERY REWRITE AS
SELECT country_id, cust_state_province, cust_city,
    prod_category, prod_subcategory, prod_name,
    GROUPING_ID(country_id, cust_state_province, cust_city,
                prod_category, prod_subcategory, prod_name) gid,
    SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY ROLLUP(country_id, cust_state_province, cust_city),
         ROLLUP(prod_category, prod_subcategory, prod_name),
PARTITION BY LIST (gid)
...;
```

This allows use of PCT refresh on the materialized views `sales_hierarchical_mon_cube_mv`, `sales_hierarchical_qtr_cube_mv`, and `sales_hierarchical_yr_cube_mv` on partition maintenance operations to sales table. PCT refresh can also be used when there have been significant changes to the base table and log based fast refresh is estimated to be slower than PCT refresh. You can just specify the method as force (`method => '?'`) in to refresh sub-programs in the `DBMS_MVIEW` package and Oracle Database will pick the best method of refresh. See "Partition Change Tracking (PCT) Refresh" on page 16-11 for more information regarding PCT refresh.

Because `sales_hierarchical_qtr_cube_mv` does not contain any column from `times` table, PCT refresh is not enabled on it. But, you can still call refresh sub-programs in the `DBMS_MVIEW` package with method as force (`method => '?'`) and Oracle Database will pick the best method of refresh.

If you are interested in a partial cube (that is, a subset of groupings from the complete cube), then Oracle recommends storing the cube as a "federated cube". A federated cube stores each grouping of interest in a separate materialized view.

```
CREATE MATERIALIZED VIEW sales_mon_city_prod_mv
PARTITION BY RANGE (mon)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_month_desc mon, cust_city, prod_name, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id
AND s.time_id = t.time_id
GROUP BY calendar_month_desc, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_qtr_city_prod_mv
PARTITION BY RANGE (qtr)
```

```
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_quarter_desc qtr, cust_city, prod_name,SUM(amount_sold) s_sales,
COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_quarter_desc, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_yr_city_prod_mv
PARTITION BY RANGE (yr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, cust_city, prod_name, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, cust_city, prod_name;

CREATE MATERIALIZED VIEW sales_mon_city_scat_mv
PARTITION BY RANGE (mon)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_month_desc mon, cust_city, prod_subcategory,
       SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id =t.time_id
GROUP BY calendar_month_desc, cust_city, prod_subcategory;

CREATE MATERIALIZED VIEW sales_qtr_city_cat_mv
PARTITION BY RANGE (qtr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_quarter_desc qtr, cust_city, prod_category cat,
       SUM(amount_sold) s_sales, COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
WHERE s.cust_id = c.cust_id AND s.prod_id =p.prod_id AND s.time_id =t.time_id
GROUP BY calendar_quarter_desc, cust_city, prod_category;

CREATE MATERIALIZED VIEW sales_yr_city_all_mv
PARTITION BY RANGE (yr)
...
BUILD DEFERRED
REFRESH FAST ON DEMAND
  USING TRUSTED CONSTRAINTS
ENABLE QUERY REWRITE AS
SELECT calendar_year yr, cust_city, SUM(amount_sold) s_sales,
       COUNT(amount_sold) c_sales, COUNT(*) c_star
FROM sales s, products p, customers c, times t
```

```
WHERE s.cust_id = c.cust_id AND s.prod_id = p.prod_id AND s.time_id = t.time_id
GROUP BY calendar_year, cust_city;
```

These materialized views can be created as `BUILD DEFERRED` and then, you can execute `DBMS_MVIEW.REFRESH_DEPENDENT(number_of_failures, 'SALES', 'C' ...)` so that the complete refresh of each of the materialized views defined on the detail table `sales` is scheduled in the most efficient order. See "Scheduling Refresh" on page 16-18 for more information.

Because each of these materialized views is partitioned on the time level (month, quarter, or year) present in the `SELECT` list, PCT is enabled on `sales` table for each one of them, thus providing an opportunity to apply PCT refresh method in addition to `FAST` and `COMPLETE` refresh methods.

22

SQL for Analysis and Reporting

The following topics provide information about how to improve analytical SQL queries in a data warehouse:

Overview of SQL for Analysis and Reporting

Oracle has enhanced SQL's analytical processing capabilities by introducing a new family of analytic SQL functions. These analytic functions enable you to calculate:

- Rankings and percentiles
- Moving window calculations
- Lag/lead analysis
- First/last analysis
- Linear regression statistics

Ranking functions include cumulative distributions, percent rank, and N-tiles. Moving window calculations allow you to find moving and cumulative aggregations, such as sums and averages. Lag/lead analysis enables direct inter-row references so you can calculate period-to-period changes. First/last analysis enables you to find the first or last value in an ordered group.

Other enhancements to SQL include the `CASE` expression and partitioned outer join. `CASE` expressions provide if-then logic useful in many situations. Partitioned outer join is an extension to ANSI outer join syntax that allows users to selectively densify certain dimensions while keeping others sparse. This allows reporting tools to selectively densify dimensions, for example, the ones that appear in their cross-tabular reports while keeping others sparse.

To enhance performance, analytic functions can be parallelized: multiple processes can simultaneously execute all of these statements. These capabilities make calculations easier and more efficient, thereby enhancing database performance, scalability, and simplicity.

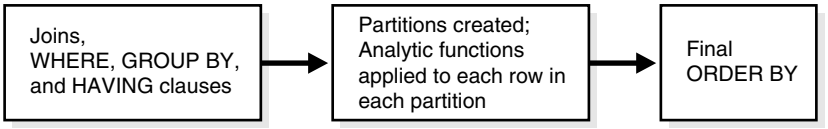
Analytic functions are classified as described in Table 22–1.

Table 22–1 Analytic Functions and Their Uses

Type	Used For
Ranking	Calculating ranks, percentiles, and n-tiles of the values in a result set.
Windowing	Calculating cumulative and moving aggregates. Works with these functions: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, `VARIANCE`, `STDDEV`, `FIRST_VALUE`, `LAST_VALUE`, and new statistical functions. Note that the `DISTINCT` keyword is not supported in windowing functions except for `MAX` and `MIN`.
Reporting	Calculating shares, for example, market share. Works with these functions: `SUM`, `AVG`, `MIN`, `MAX`, `COUNT` (with/without `DISTINCT`), `VARIANCE`, `STDDEV`, `RATIO_TO_REPORT`, and new statistical functions. Note that the `DISTINCT` keyword may be used in those reporting functions that support `DISTINCT` in aggregate mode.
`LAG/LEAD`	Finding a value in a row a specified number of rows from a current row.
`FIRST/LAST`	First or last value in an ordered group.
Linear Regression	Calculating linear regression and other statistics (slope, intercept, and so on).
Inverse Percentile	The value in a data set that corresponds to a specified percentile.
Hypothetical Rank and Distribution	The rank or percentile that a row would have if inserted into a specified data set.

To perform these operations, the analytic functions add several new elements to SQL processing. These elements build on existing SQL to allow flexible and powerful calculation expressions. With just a few exceptions, the analytic functions have these new elements. The processing flow is represented in Figure 22–1.

Figure 22–1 Processing Order



The essential concepts used in analytic functions are:

- Processing order

  Query processing using analytic functions takes place in three stages. First, all joins, `WHERE`, `GROUP BY` and `HAVING` clauses are performed. Second, the result set is made available to the analytic functions, and all their calculations take place. Third, if the query has an `ORDER BY` clause at its end, the `ORDER BY` is processed to allow for precise output ordering. The processing order is shown in Figure 22–1.

- Result set partitions

  The analytic functions allow users to divide query result sets into groups of rows called partitions. Note that the term **partitions** used with analytic functions is unrelated to the table partitions feature. Throughout this chapter, the term partitions refers to only the meaning related to analytic functions. Partitions are created after the groups defined with `GROUP BY` clauses, so they are available to any aggregate results such as sums and averages. Partition divisions may be based upon any desired columns or expressions. A query result set may be partitioned into just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each.

- Window

For each row in a partition, you can define a sliding window of data. This window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time. The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends. For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its partition, and its ending row would slide from the starting point all the way to the last row of the partition. In contrast, a window defined for a moving average would have both its starting and end points slide so that they maintain a constant physical or logical range.

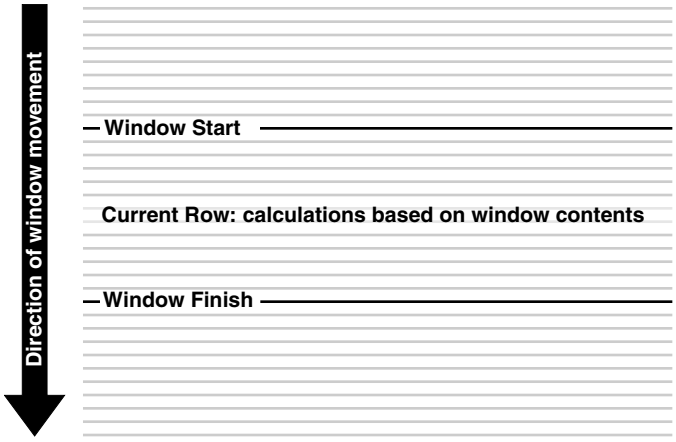
A window can be set as large as all the rows in a partition or just a sliding window of one row within a partition. When a window is near a border, the function returns results for only the available rows, rather than warning you that the results are not what you want.

When using window functions, the current row is included during calculations, so you should only specify (*n*-1) when you are dealing with *n* items.

- Current row

  Each calculation performed with an analytic function is based on a current row within a partition. The current row serves as the reference point determining the start and end of the window. For instance, a centered moving average calculation could be defined with a window that holds the current row, the six preceding rows, and the following six rows. This would create a sliding window of 13 rows, as shown in Figure 22–2.

Figure 22–2 Sliding Window Example



Ranking, Windowing, and Reporting Functions

This section illustrates the basic analytic functions for ranking, windowing, and reporting.

Ranking

A ranking function computes the rank of a record compared to other records in the data set based on the values of a set of measures. The types of ranking function are:

RANK and DENSE_RANK Functions

The RANK and DENSE_RANK functions allow you to rank items in a group, for example, finding the top three products sold in California last year. There are two functions that perform ranking, as shown by the following syntax:

```
RANK ( ) OVER ( [query_partition_clause] order_by_clause )
DENSE_RANK ( ) OVER ( [query_partition_clause] order_by_clause )
```

The difference between RANK and DENSE_RANK is that DENSE_RANK leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using DENSE_RANK and had three people tie for second place, you would say that all three were in second place and that the next person came in third. The RANK function would also give three people in second place, but the next person would be in fifth place.

The following are some relevant points about RANK:

- Ascending is the default sort order, which you may want to change to descending.
- The expressions in the optional PARTITION BY clause divide the query result set into groups within which the RANK function operates. That is, RANK gets reset whenever the group changes. In effect, the value expressions of the PARTITION BY clause define the reset boundaries.
- If the PARTITION BY clause is missing, then ranks are computed over the entire query result set.
- The ORDER BY clause specifies the measures (<value expression>) on which ranking is done and defines the order in which rows are sorted in each group (or partition). Once the data is sorted within each partition, ranks are given to each row starting from 1.
- The NULLS FIRST | NULLS LAST clause indicates the position of NULLs in the ordered sequence, either first or last in the sequence. The order of the sequence would make NULLs compare either high or low with respect to non-NULL values. If the sequence were in ascending order, then NULLS FIRST implies that NULLs are smaller than all other non-NULL values and NULLS LAST implies they are larger than non-NULL values. It is the opposite for descending order. See the example in "Treatment of NULLs" on page 22-7.
- If the NULLS FIRST | NULLS LAST clause is omitted, then the ordering of the null values depends on the ASC or DESC arguments. Null values are considered larger than any other values. If the ordering sequence is ASC, then nulls will appear last; nulls will appear first otherwise. Nulls are considered equal to other nulls and, therefore, the order in which nulls are presented is non-deterministic.

Ranking Order The following example shows how the [ASC | DESC] option changes the ranking order.

Example 22–1 Ranking Order

```
SELECT channel_desc, TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
   RANK() OVER (ORDER BY SUM(amount_sold)) AS default_rank,
   RANK() OVER (ORDER BY SUM(amount_sold) DESC NULLS LAST) AS custom_rank
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id AND sales.time_id=times.time_id
  AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND country_iso_code='US'
GROUP BY channel_desc;
```

```
CHANNEL_DESC         SALES$         DEFAULT_RANK CUSTOM_RANK
-------------------- -------------- ------------ -----------
Direct Sales              1,320,497            3           1
Partners                    800,871            2           2
Internet                    261,278            1           3
```

While the data in this result is ordered on the measure SALES$, in general, it is not guaranteed by the RANK function that the data will be sorted on the measures. If you want the data to be sorted on SALES$ in your result, you must specify it explicitly with an ORDER BY clause, at the end of the SELECT statement.

Ranking on Multiple Expressions Ranking functions must resolve ties between values in the set. If the first expression cannot resolve ties, the second expression is used to resolve ties and so on. For example, here is a query ranking three of the sales channels over two months based on their dollar sales, breaking ties with the unit sales. (Note that the TRUNC function is used here only to create tie values for this query.)

Example 22–2 Ranking On Multiple Expressions

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(TRUNC(SUM(amount_sold),-5),
  '9,999,999,999') SALES$, TO_CHAR(SUM(quantity_sold), '9,999,999,999')
  SALES_Count, RANK() OVER (ORDER BY TRUNC(SUM(amount_sold), -5)
  DESC, SUM(quantity_sold) DESC) AS col_rank
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         SALES_COUNT     COL_RANK
-------------------- -------- -------------- -------------- ---------
Direct Sales         2000-10       1,200,000         12,584         1
Direct Sales         2000-09       1,200,000         11,995         2
Partners             2000-10         600,000          7,508         3
Partners             2000-09         600,000          6,165         4
Internet             2000-09         200,000          1,887         5
Internet             2000-10         200,000          1,450         6
```

The sales_count column breaks the ties for three pairs of values.

RANK and DENSE_RANK Difference The difference between RANK and DENSE_RANK functions is illustrated in Example 22–3.

Example 22–3 RANK and DENSE_RANK

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold),-5), '9,999,999,999') SALES$,
      RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-5) DESC) AS RANK,
DENSE_RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-5) DESC) AS DENSE_RANK
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id
  AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-10')
  AND channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;
```

```
CHANNEL_DESC         CALENDAR SALES$              RANK DENSE_RANK
-------------------- -------- -------------- --------- ----------
Direct Sales         2000-09       1,200,000         1          1
Direct Sales         2000-10       1,200,000         1          1
Partners             2000-09         600,000         3          2
Partners             2000-10         600,000         3          2
Internet             2000-09         200,000         5          3
Internet             2000-10         200,000         5          3
```

Note that, in the case of DENSE_RANK, the largest rank value gives the number of distinct values in the data set.

Per Group Ranking The RANK function can be made to operate within groups, that is, the rank gets reset whenever the group changes. This is accomplished with the PARTITION BY clause. The group expressions in the PARTITION BY subclause divide the data set into groups within which RANK operates. For example, to rank products within each channel by their dollar sales, you could issue the following statement.

Example 22–4 Per Group Ranking Example 1

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
 '9,999,999,999') SALES$, RANK() OVER (PARTITION BY channel_desc
  ORDER BY SUM(amount_sold) DESC) AS RANK_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         RANK_BY_CHANNEL
-------------------- -------- -------------- ---------------
Direct Sales         2000-08       1,236,104               1
Direct Sales         2000-10       1,225,584               2
Direct Sales         2000-09       1,217,808               3
Direct Sales         2000-11       1,115,239               4
Internet             2000-11         284,742               1
Internet             2000-10         239,236               2
Internet             2000-09         228,241               3
Internet             2000-08         215,107               4

8 rows selected.
```

A single query block can contain more than one ranking function, each partitioning the data into different groups (that is, reset on different boundaries). The groups can be mutually exclusive. The following query ranks products based on their dollar sales within each month (rank_of_product_per_region) and within each channel (rank_of_product_total).

Example 22–5 Per Group Ranking Example 2

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
  '9,999,999,999') SALES$, RANK() OVER (PARTITION BY calendar_month_desc
  ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_MONTH, RANK() OVER (PARTITION
 BY channel_desc ORDER BY SUM(amount_sold) DESC) AS RANK_WITHIN_CHANNEL
FROM sales, products, customers, times, channels, countries
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND customers.country_id = countries.country_id AND sales.time_id=times.time_id
  AND sales.channel_id=channels.channel_id
```

```
  AND times.calendar_month_desc IN ('2000-08', '2000-09', '2000-10', '2000-11')
  AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC  CALENDAR    SALES$     RANK_WITHIN_MONTH  RANK_WITHIN_CHANNEL
------------- --------    ---------  -----------------  -------------------
Direct Sales   2000-08    1,236,104                  1                    1
Internet       2000-08      215,107                  2                    4
Direct Sales   2000-09    1,217,808                  1                    3
Internet       2000-09      228,241                  2                    3
Direct Sales   2000-10    1,225,584                  1                    2
Internet       2000-10      239,236                  2                    2
Direct Sales   2000-11    1,115,239                  1                    4
Internet       2000-11      284,742                  2                    1
```

Per Cube and Rollup Group Ranking Analytic functions, RANK for example, can be reset based on the groupings provided by a CUBE, ROLLUP, or GROUPING SETS operator. It is useful to assign ranks to the groups created by CUBE, ROLLUP, and GROUPING SETS queries. See Chapter 21, "SQL for Aggregation in Data Warehouses" for further information about the GROUPING function.

A sample CUBE and ROLLUP query is the following:

```
SELECT channel_desc, country_iso_code,
  TO_CHAR(SUM(amount_sold), '9,999,999,999')
 SALES$, RANK() OVER (PARTITION BY GROUPING_ID(channel_desc, country_iso_code)
    ORDER BY SUM(amount_sold) DESC) AS RANK_PER_GROUP
FROM sales, customers, times, channels, countries
WHERE sales.time_id=times.time_id AND sales.cust_id=customers.cust_id
  AND sales.channel_id = channels.channel_id AND channels.channel_desc
   IN ('Direct Sales', 'Internet') AND times.calendar_month_desc='2000-09'
   AND country_iso_code IN ('GB', 'US', 'JP')
GROUP BY CUBE(channel_desc, country_iso_code);

CHANNEL_DESC         CO SALES$         RANK_PER_GROUP
-------------------- -- -------------- --------------
Direct Sales         GB      1,217,808              1
Direct Sales         JP      1,217,808              1
Direct Sales         US      1,217,808              1
Internet             GB        228,241              4
Internet             JP        228,241              4
Internet             US        228,241              4
Direct Sales                 3,653,423              1
Internet                       684,724              2
                     GB      1,446,049              1
                     JP      1,446,049              1
                     US      1,446,049              1
                             4,338,147              1
```

Treatment of NULLs NULLs are treated like normal values. Also, for rank computation, a NULL value is assumed to be equal to another NULL value. Depending on the ASC | DESC options provided for measures and the NULLS FIRST | NULLS LAST clause, NULLs will either sort low or high and hence, are given ranks appropriately. The following example shows how NULLs are ranked in different cases:

```
SELECT  times.time_id time,  sold,
  RANK() OVER (ORDER BY (sold) DESC NULLS LAST) AS NLAST_DESC,
  RANK() OVER (ORDER BY (sold) DESC NULLS FIRST) AS NFIRST_DESC,
  RANK() OVER (ORDER BY (sold) ASC NULLS FIRST) AS NFIRST,
  RANK() OVER (ORDER BY (sold) ASC NULLS LAST) AS NLAST
```

```
FROM
  (
   SELECT time_id, SUM(sales.amount_sold)  sold
   FROM sales, products, customers, countries
   WHERE sales.prod_id=products.prod_id
    AND customers.country_id = countries.country_id
    AND sales.cust_id=customers.cust_id
    AND prod_name IN ('Envoy Ambassador', 'Mouse Pad') AND country_iso_code ='GB'
   GROUP BY  time_id)
 v, times
WHERE v.time_id (+) = times.time_id
  AND calendar_year=1999
  AND calendar_month_number=1
ORDER BY sold  DESC NULLS LAST;

TIME            SOLD NLAST_DESC NFIRST_DESC     NFIRST      NLAST
--------- ---------- ---------- ------------ ---------- ----------
25-JAN-99    3097.32          1           18         31         14
17-JAN-99    1791.77          2           19         30         13
30-JAN-99     127.69          3           20         29         12
28-JAN-99     120.34          4           21         28         11
23-JAN-99      86.12          5           22         27         10
20-JAN-99      79.07          6           23         26          9
13-JAN-99       56.1          7           24         25          8
07-JAN-99      42.97          8           25         24          7
08-JAN-99      33.81          9           26         23          6
10-JAN-99      22.76         10           27         21          4
02-JAN-99      22.76         10           27         21          4
26-JAN-99      19.84         12           29         20          3
16-JAN-99      11.27         13           30         19          2
14-JAN-99       9.52         14           31         18          1
09-JAN-99                    15            1          1         15
12-JAN-99                    15            1          1         15
31-JAN-99                    15            1          1         15
11-JAN-99                    15            1          1         15
19-JAN-99                    15            1          1         15
03-JAN-99                    15            1          1         15
15-JAN-99                    15            1          1         15
21-JAN-99                    15            1          1         15
24-JAN-99                    15            1          1         15
04-JAN-99                    15            1          1         15
06-JAN-99                    15            1          1         15
27-JAN-99                    15            1          1         15
18-JAN-99                    15            1          1         15
01-JAN-99                    15            1          1         15
22-JAN-99                    15            1          1         15
29-JAN-99                    15            1          1         15
05-JAN-99                    15            1          1         15
```

Bottom N Ranking

Bottom N is similar to top N except for the ordering sequence within the rank expression. Using the previous example, you can order `SUM(s_amount)` ascending instead of descending.

CUME_DIST Function

The `CUME_DIST` function (defined as the inverse of percentile in some statistical books) computes the position of a specified value relative to a set of values. The order can be ascending or descending. Ascending is the default. The range of values for

CUME_DIST is from greater than 0 to 1. To compute the CUME_DIST of a value x in a set S of size N, you use the formula:

```
CUME_DIST(x) =  number of values in S coming before
   and including x in the specified order/ N
```

Its syntax is:

```
CUME_DIST ( ) OVER ( [query_partition_clause] order_by_clause )
```

The semantics of various options in the CUME_DIST function are similar to those in the RANK function. The default order is ascending, implying that the lowest value gets the lowest CUME_DIST (as all other values come later than this value in the order). NULLs are treated the same as they are in the RANK function. They are counted toward both the numerator and the denominator as they are treated like non-NULL values. The following example finds cumulative distribution of sales by channel within each month:

```
SELECT calendar_month_desc AS MONTH, channel_desc,
     TO_CHAR(SUM(amount_sold) , '9,999,999,999') SALES$,
     CUME_DIST() OVER (PARTITION BY calendar_month_desc ORDER BY
         SUM(amount_sold) ) AS CUME_DIST_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2000-09', '2000-07','2000-08')
GROUP BY calendar_month_desc, channel_desc;

MONTH    CHANNEL_DESC         SALES$         CUME_DIST_BY_CHANNEL
-------- -------------------- -------------- --------------------
2000-07  Internet                140,423           .333333333
2000-07  Partners                611,064           .666666667
2000-07  Direct Sales          1,145,275                    1
2000-08  Internet                215,107           .333333333
2000-08  Partners                661,045           .666666667
2000-08  Direct Sales          1,236,104                    1
2000-09  Internet                228,241           .333333333
2000-09  Partners                666,172           .666666667
2000-09  Direct Sales          1,217,808                    1
```

PERCENT_RANK Function

PERCENT_RANK is similar to CUME_DIST, but it uses rank values rather than row counts in its numerator. Therefore, it returns the percent rank of a value relative to a group of values. The function is available in many popular spreadsheets. PERCENT_RANK of a row is calculated as:

```
(rank of row in its partition - 1) / (number of rows in the partition - 1)
```

PERCENT_RANK returns values in the range zero to one. The row(s) with a rank of 1 will have a PERCENT_RANK of zero. Its syntax is:

```
PERCENT_RANK () OVER ([query_partition_clause] order_by_clause)
```

NTILE Function

NTILE allows easy calculation of tertiles, quartiles, deciles and other common summary statistics. This function divides an ordered partition into a specified number of groups called **buckets** and assigns a bucket number to each row in the partition. NTILE is a very useful calculation because it lets users divide a data set into fourths, thirds, and other groupings.

The buckets are calculated so that each bucket has exactly the same number of rows assigned to it or at most 1 row more than the others. For instance, if you have 100 rows in a partition and ask for an NTILE function with four buckets, 25 rows will be assigned a value of 1, 25 rows will have value 2, and so on. These buckets are referred to as equiheight buckets.

If the number of rows in the partition does not divide evenly (without a remainder) into the number of buckets, then the number of rows assigned for each bucket will differ by one at most. The extra rows will be distributed one for each bucket starting from the lowest bucket number. For instance, if there are 103 rows in a partition which has an NTILE(5) function, the first 21 rows will be in the first bucket, the next 21 in the second bucket, the next 21 in the third bucket, the next 20 in the fourth bucket and the final 20 in the fifth bucket.

The NTILE function has the following syntax:

```
NTILE (expr) OVER ([query_partition_clause] order_by_clause)
```

In this, the N in NTILE(N) can be a constant (for example, 5) or an expression.

This function, like RANK and CUME_DIST, has a PARTITION BY clause for per group computation, an ORDER BY clause for specifying the measures and their sort order, and NULLS FIRST | NULLS LAST clause for the specific treatment of NULLs. For example, the following is an example assigning each month's sales total into one of four buckets:

```
SELECT calendar_month_desc AS MONTH , TO_CHAR(SUM(amount_sold),
 '9,999,999,999')
 SALES$, NTILE(4) OVER (ORDER BY SUM(amount_sold)) AS TILE4
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_year=2000 AND prod_category= 'Electronics'
GROUP BY calendar_month_desc;

MONTH    SALES$              TILE4
-------- -------------- ----------
2000-02         242,416          1
2000-01         257,286          1
2000-03         280,011          1
2000-06         315,951          2
2000-05         316,824          2
2000-04         318,106          2
2000-07         433,824          3
2000-08         477,833          3
2000-12         553,534          3
2000-10         652,225          4
2000-11         661,147          4
2000-09         691,449          4
```

NTILE ORDER BY statements must be fully specified to yield reproducible results. Equal values can get distributed across adjacent buckets. To ensure deterministic results, you must order on a unique key.

ROW_NUMBER Function

The ROW_NUMBER function assigns a unique number (sequentially, starting from 1, as defined by ORDER BY) to each row within the partition. It has the following syntax:

```
ROW_NUMBER ( ) OVER ( [query_partition_clause] order_by_clause )
```

Example 22–6 ROW_NUMBER

```
SELECT channel_desc, calendar_month_desc,
   TO_CHAR(TRUNC(SUM(amount_sold), -5), '9,999,999,999') SALES$,
   ROW_NUMBER() OVER (ORDER BY TRUNC(SUM(amount_sold), -6) DESC) AS ROW_NUMBER
FROM sales, products, customers, times, channels
WHERE sales.prod_id=products.prod_id AND sales.cust_id=customers.cust_id
  AND sales.time_id=times.time_id AND sales.channel_id=channels.channel_id
  AND times.calendar_month_desc IN ('2001-09', '2001-10')
GROUP BY channel_desc, calendar_month_desc;

CHANNEL_DESC         CALENDAR SALES$         ROW_NUMBER
-------------------- -------- -------------- ----------
Direct Sales         2001-10       1,000,000          1
Direct Sales         2001-09       1,100,000          2
Internet             2001-09         500,000          3
Partners             2001-09         600,000          4
Partners             2001-10         600,000          5
Internet             2001-10         700,000          6
```

Note that there are three pairs of tie values in these results. Like NTILE, ROW_NUMBER is a non-deterministic function, so each tied value could have its row number switched. To ensure deterministic results, you must order on a unique key. Inmost cases, that will require adding a new tie breaker column to the query and using it in the ORDER BY specification.

Windowing

Windowing functions can be used to compute cumulative, moving, and centered aggregates. They return a value for each row in the table, which depends on other rows in the corresponding window. With windowing aggregate functions, you can calculate moving and cumulative versions of SUM, AVERAGE, COUNT, MAX, MIN, and many more functions. They can be used only in the SELECT and ORDER BY clauses of the query. Windowing aggregate functions include the convenient FIRST_VALUE, which returns the first value in the window; and LAST_VALUE, which returns the last value in the window. These functions provide access to more than one row of a table without a self-join. The syntax of the windowing functions is:

```
analytic_function([ arguments ])
   OVER (analytic_clause)

 where analytic_clause =
     [ query_partition_clause ]
     [ order_by_clause [ windowing_clause ] ]

and query_partition_clause =
    PARTITION BY
      { value_expr[, value_expr ]...
      }

 and windowing_clause =
     { ROWS | RANGE }
     { BETWEEN
       { UNBOUNDED PRECEDING
     | CURRENT ROW
     | value_expr { PRECEDING | FOLLOWING }
     }
     AND
     { UNBOUNDED FOLLOWING
     | CURRENT ROW
```

```
    | value_expr { PRECEDING | FOLLOWING }
    }
  | { UNBOUNDED PRECEDING
    | CURRENT ROW
    | value_expr PRECEDING
    }
  }
```

Note that the DISTINCT keyword is not supported in windowing functions except for MAX and MIN.

> **See Also:** *Oracle Database SQL Language Reference* for further information regarding syntax and restrictions

Treatment of NULLs as Input to Window Functions

Window functions' NULL semantics match the NULL semantics for SQL aggregate functions. Other semantics can be obtained by user-defined functions, or by using the DECODE or a CASE expression within the window function.

Windowing Functions with Logical Offset

A logical offset can be specified with constants such as RANGE 10 PRECEDING, or an expression that evaluates to a constant, or by an interval specification like RANGE INTERVAL N DAY/MONTH/YEAR PRECEDING or an expression that evaluates to an interval.

With logical offset, there can only be one expression in the ORDER BY expression list in the function, with type compatible to NUMERIC if offset is numeric, or DATE if an interval is specified.

An analytic function that uses the RANGE keyword can use multiple sort keys in its ORDER BY clause if it specifies either of these two windows:

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. The short form of this is RANGE UNBOUNDED PRECEDING, which can also be used.
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

Window boundaries that do not meet these conditions can have only one sort key in the analytic function's ORDER BY clause.

Example 22–7 Cumulative Aggregate Function

The following is an example of cumulative amount_sold by customer ID by quarter in 2000:

```
SELECT c.cust_id, t.calendar_quarter_desc, TO_CHAR (SUM(amount_sold),
  '9,999,999,999.99') AS Q_SALES, TO_CHAR(SUM(SUM(amount_sold))
OVER (PARTITION BY c.cust_id ORDER BY c.cust_id, t.calendar_quarter_desc
ROWS UNBOUNDED
PRECEDING), '9,999,999,999.99') AS CUM_SALES
  FROM sales s, times t, customers c
  WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND t.calendar_year=2000
    AND c.cust_id IN (2595, 9646, 11111)
  GROUP BY c.cust_id, t.calendar_quarter_desc
  ORDER BY c.cust_id, t.calendar_quarter_desc;

   CUST_ID CALENDA Q_SALES           CUM_SALES
---------- ------- ----------------- -----------------
      2595 2000-01            659.92            659.92
      2595 2000-02            224.79            884.71
```

```
      2595 2000-03            313.90          1,198.61
      2595 2000-04          6,015.08          7,213.69
      9646 2000-01          1,337.09          1,337.09
      9646 2000-02            185.67          1,522.76
      9646 2000-03            203.86          1,726.62
      9646 2000-04            458.29          2,184.91
     11111 2000-01             43.18             43.18
     11111 2000-02             33.33             76.51
     11111 2000-03            579.73            656.24
     11111 2000-04            307.58            963.82
```

In this example, the analytic function SUM defines, for each row, a window that starts at the beginning of the partition (UNBOUNDED PRECEDING) and ends, by default, at the current row.

Nested SUMs are needed in this example since we are performing a SUM over a value that is itself a SUM. Nested aggregations are used very often in analytic aggregate functions.

Example 22–8 Moving Aggregate Function

This example of a time-based window shows, for one customer, the moving average of sales for the current month and preceding two months:

```
SELECT c.cust_id, t.calendar_month_desc, TO_CHAR (SUM(amount_sold),
      '9,999,999,999') AS SALES, TO_CHAR(AVG(SUM(amount_sold))
OVER (ORDER BY c.cust_id, t.calendar_month_desc ROWS 2 PRECEDING),
      '9,999,999,999') AS MOVING_3_MONTH_AVG
FROM sales s, times t, customers c
WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id
   AND t.calendar_year=1999 AND c.cust_id IN (6510)
GROUP BY c.cust_id, t.calendar_month_desc
ORDER BY c.cust_id, t.calendar_month_desc;

   CUST_ID CALENDAR SALES          MOVING_3_MONTH
---------- -------- -------------- --------------
      6510 1999-04            125            125
      6510 1999-05          3,395          1,760
      6510 1999-06          4,080          2,533
      6510 1999-07          6,435          4,637
      6510 1999-08          5,105          5,207
      6510 1999-09          4,676          5,405
      6510 1999-10          5,109          4,963
      6510 1999-11            802          3,529
```

Note that the first two rows for the three month moving average calculation in the output data are based on a smaller interval size than specified because the window calculation cannot reach past the data retrieved by the query. You must consider the different window sizes found at the borders of result sets. In other words, you may need to modify the query to include exactly what you want.

Centered Aggregate Function

Calculating windowing aggregate functions centered around the current row is straightforward. This example computes for all customers a centered moving average of sales for one week in late December 1999. It finds an average of the sales total for the one day preceding the current row and one day following the current row including the current row as well.

Example 22–9 Centered Aggregate

```
SELECT t.time_id, TO_CHAR (SUM(amount_sold), '9,999,999,999')
AS SALES, TO_CHAR(AVG(SUM(amount_sold)) OVER
  (ORDER BY t.time_id
   RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND
   INTERVAL '1' DAY FOLLOWING), '9,999,999,999') AS CENTERED_3_DAY_AVG
   FROM sales s, times t
   WHERE s.time_id=t.time_id AND t.calendar_week_number IN (51)
     AND calendar_year=1999
   GROUP BY t.time_id
   ORDER BY t.time_id;

TIME_ID   SALES          CENTERED_3_DAY
--------- -------------- --------------
20-DEC-99        134,337        106,676
21-DEC-99         79,015        102,539
22-DEC-99         94,264         85,342
23-DEC-99         82,746         93,322
24-DEC-99        102,957         82,937
25-DEC-99         63,107         87,062
26-DEC-99         95,123         79,115
```

The starting and ending rows for each product's centered moving average calculation in the output data are based on just two days, since the window calculation cannot reach past the data retrieved by the query. Users must consider the different window sizes found at the borders of result sets: the query may need to be adjusted.

Windowing Aggregate Functions in the Presence of Duplicates

The following example illustrates how window aggregate functions compute values when there are duplicates, that is, when multiple rows are returned for a single ordering value. The query retrieves the quantity sold to several customers during a specified time range. (Although we use an inline view to define our base data set, it has no special significance and can be ignored.) The query defines a moving window that runs from the date of the current row to 10 days earlier.

Note that the `RANGE` keyword is used to define the windowing clause of this example. This means that the window can potentially hold many rows for each value in the range. In this case, there are three pairs of rows with duplicate date values.

Example 22–10 Windowing Aggregate Functions with Logical Offsets

```
SELECT time_id, daily_sum, SUM(daily_sum) OVER (ORDER BY time_id
RANGE BETWEEN INTERVAL '10' DAY PRECEDING AND CURRENT ROW)
AS current_group_sum
FROM (SELECT time_id, channel_id, SUM(s.quantity_sold)
AS daily_sum
FROM customers c, sales s, countries
WHERE c.cust_id=s.cust_id
  AND c.country_id = countries.country_id
  AND s.cust_id IN (638, 634, 753, 440 ) AND s.time_id BETWEEN '01-MAY-00'
  AND '13-MAY-00' GROUP BY time_id, channel_id);

TIME_ID    DAILY_SUM CURRENT_GROUP_SUM
--------- ---------- -----------------
06-MAY-00          7                 7   /* 7 */
10-MAY-00          1                 9   /* 7 + (1+1) */
10-MAY-00          1                 9   /* 7 + (1+1) */
11-MAY-00          2                15   /* 7 + (1+1) + (2+4) */
11-MAY-00          4                15   /* 7 + (1+1) + (2+4) */
```

```
12-MAY-00          1                16   /* 7 + (1+1) + (2+4) + 1 */
13-MAY-00          2                23   /* 7 + (1+1) + (2+4) + 1 + (5+2) */
13-MAY-00          5                23   /* 7 + (1+1) + (2+4) + 1 + (5+2) */
```

In the output of this example, all dates except May 6 and May 12 return two rows. Examine the commented numbers to the right of the output to see how the values are calculated. Note that each group in parentheses represents the values returned for a single day.

Note that this example applies only when you use the RANGE keyword rather than the ROWS keyword. It is also important to remember that with RANGE, you can only use 1 ORDER BY expression in the analytic function's ORDER BY clause. With the ROWS keyword, you can use multiple order by expressions in the analytic function's ORDER BY clause.

Varying Window Size for Each Row

There are situations where it is useful to vary the size of a window for each row, based on a specified condition. For instance, you may want to make the window larger for certain dates and smaller for others. Assume that you want to calculate the moving average of stock price over three working days. If you have an equal number of rows for each day for all working days and no non-working days are stored, then you can use a physical window function. However, if the conditions noted are not met, you can still calculate a moving average by using an expression in the window size parameters.

Expressions in a window size specification can be made in several different sources. the expression could be a reference to a column in a table, such as a time table. It could also be a function that returns the appropriate boundary for the window based on values in the current row. The following statement for a hypothetical stock price database uses a user-defined function in its RANGE clause to set window size:

```
SELECT t_timekey, AVG(stock_price)
        OVER (ORDER BY t_timekey RANGE fn(t_timekey) PRECEDING) av_price
FROM stock, time WHERE st_timekey = t_timekey
ORDER BY t_timekey;
```

In this statement, t_timekey is a date field. Here, *fn* could be a PL/SQL function with the following specification:

fn(t_timekey) returns

- 4 if t_timekey is Monday, Tuesday
- 2 otherwise
- If any of the previous days are holidays, it adjusts the count appropriately.

Note that, when window is specified using a number in a window function with ORDER BY on a date column, then it is converted to mean the number of days. You could have also used the interval literal conversion function, as NUMTODSINTERVAL(fn(t_timekey), 'DAY') instead of just fn(t_timekey) to mean the same thing. You can also write a PL/SQL function that returns an INTERVAL data type value.

Windowing Aggregate Functions with Physical Offsets

For windows expressed in rows, the ordering expressions should be unique to produce deterministic results. For example, the following query is not deterministic because time_id is not unique in this result set.

Example 22–11 Windowing Aggregate Functions With Physical Offsets

```
SELECT t.time_id, TO_CHAR(amount_sold, '9,999,999,999') AS INDIV_SALE,
       TO_CHAR(SUM(amount_sold) OVER (PARTITION BY t.time_id ORDER BY t.time_id
ROWS UNBOUNDED PRECEDING), '9,999,999,999') AS CUM_SALES
FROM sales s, times t, customers c
WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id
  AND t.time_id IN
   (TO_DATE('11-DEC-1999'), TO_DATE('12-DEC-1999'))
  AND c.cust_id
BETWEEN 6500 AND 6600
ORDER BY t.time_id;

TIME_ID   INDIV_SALE    CUM_SALES
--------- ----------    ---------
12-DEC-99         23           23
12-DEC-99          9           32
12-DEC-99         14           46
12-DEC-99         24           70
12-DEC-99         19           89
```

One way to handle this problem would be to add the `prod_id` column to the result set and order on both `time_id` and `prod_id`.

Reporting

After a query has been processed, aggregate values like the number of resulting rows or an average value in a column can be easily computed within a partition and made available to other reporting functions. Reporting aggregate functions return the same aggregate value for every row in a partition. Their behavior with respect to `NULL`s is the same as the SQL aggregate functions. The syntax is:

```
{SUM | AVG | MAX | MIN | COUNT | STDDEV | VARIANCE  ... }
  ([ALL | DISTINCT] {value expression1 [,...] | *})
   OVER ([PARTITION BY value expression2[,...]])
```

In addition, the following conditions apply:

- An asterisk (*) is only allowed in `COUNT(*)`
- `DISTINCT` is supported only if corresponding aggregate functions allow it.
- *`value expression1`* and *`value expression2`* can be any valid expression involving column references or aggregates.
- The `PARTITION BY` clause defines the groups on which the windowing functions would be computed. If the `PARTITION BY` clause is absent, then the function is computed over the whole query result set.

Reporting functions can appear only in the `SELECT` clause or the `ORDER BY` clause. The major benefit of reporting functions is their ability to do multiple passes of data in a single query block and speed up query performance. Queries such as "Count the number of salesmen with sales more than 10% of city sales" do not require joins between separate query blocks.

For example, consider the question "For each product category, find the region in which it had maximum sales". The equivalent SQL query using the `MAX` reporting aggregate function would be:

```
SELECT prod_category, country_region, sales
FROM (SELECT SUBSTR(p.prod_category,1,8) AS prod_category, co.country_region,
 SUM(amount_sold) AS sales,
```

```
MAX(SUM(amount_sold)) OVER (PARTITION BY prod_category) AS MAX_REG_SALES
FROM sales s, customers c, countries co, products p
WHERE s.cust_id=c.cust_id AND c.country_id=co.country_id
   AND s.prod_id =p.prod_id AND s.time_id = TO_DATE('11-OCT-2001')
GROUP BY prod_category, country_region)
WHERE sales = MAX_REG_SALES;
```

The inner query with the reporting aggregate function `MAX(SUM(amount_sold))` returns:

```
PROD_CAT COUNTRY_REGION            SALES MAX_REG_SALES
-------- -------------------- ---------- -------------
Electron Americas                581.92        581.92
Hardware Americas                925.93        925.93
Peripher Americas               3084.48       4290.38
Peripher Asia                   2616.51       4290.38
Peripher Europe                 4290.38       4290.38
Peripher Oceania                 940.43       4290.38
Software Americas                4445.7        4445.7
Software Asia                   1408.19        4445.7
Software Europe                 3288.83        4445.7
Software Oceania                 890.25        4445.7
```

The full query results are:

```
PROD_CAT COUNTRY_REGION            SALES
-------- -------------------- ----------
Electron Americas                581.92
Hardware Americas                925.93
Peripher Europe                 4290.38
Software Americas                4445.7
```

Example 22–12 Reporting Aggregate Example

Reporting aggregates combined with nested queries enable you to answer complex queries efficiently. For example, what if you want to know the best selling products in your most significant product subcategories? The following is a query which finds the 5 top-selling products for each product subcategory that contributes more than 20% of the sales within its product category:

```
SELECT SUBSTR(prod_category,1,8) AS CATEG, prod_subcategory, prod_id, SALES
FROM (SELECT p.prod_category, p.prod_subcategory, p.prod_id,
      SUM(amount_sold) AS SALES,
      SUM(SUM(amount_sold)) OVER (PARTITION BY p.prod_category) AS CAT_SALES,
      SUM(SUM(amount_sold)) OVER
         (PARTITION BY p.prod_subcategory) AS SUBCAT_SALES,
      RANK() OVER  (PARTITION BY p.prod_subcategory
         ORDER BY SUM(amount_sold) ) AS RANK_IN_LINE
     FROM sales s, customers c, countries co, products p
     WHERE s.cust_id=c.cust_id
       AND c.country_id=co.country_id AND  s.prod_id=p.prod_id
       AND s.time_id=to_DATE('11-OCT-2000')
     GROUP BY p.prod_category, p.prod_subcategory, p.prod_id
     ORDER BY prod_category, prod_subcategory)
  WHERE SUBCAT_SALES>0.2*CAT_SALES AND RANK_IN_LINE<=5;
```

RATIO_TO_REPORT Function

The `RATIO_TO_REPORT` function computes the ratio of a value to the sum of a set of values. If the expression `value expression` evaluates to `NULL`, `RATIO_TO_REPORT`

also evaluates to NULL, but it is treated as zero for computing the sum of values for the denominator. Its syntax is:

```
RATIO_TO_REPORT ( expr ) OVER ( [query_partition_clause] )
```

In this, the following applies:

- expr can be any valid expression involving column references or aggregates.
- The PARTITION BY clause defines the groups on which the RATIO_TO_REPORT function is to be computed. If the PARTITION BY clause is absent, then the function is computed over the whole query result set.

Example 22–13 RATIO_TO_REPORT

To calculate RATIO_TO_REPORT of sales for each channel, you might use the following syntax:

```
SELECT ch.channel_desc, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
     TO_CHAR(SUM(SUM(amount_sold)) OVER (), '9,999,999') AS TOTAL_SALES,
     TO_CHAR(RATIO_TO_REPORT(SUM(amount_sold)) OVER (), '9.999')
     AS RATIO_TO_REPORT
FROM sales s, channels ch
WHERE s.channel_id=ch.channel_id  AND s.time_id=to_DATE('11-OCT-2000')
GROUP BY ch.channel_desc;

CHANNEL_DESC         SALES      TOTAL_SALE RATIO_
-------------------- ---------- ---------- ------
Direct Sales             14,447     23,183   .623
Internet                    345     23,183   .015
Partners                  8,391     23,183   .362
```

LAG/LEAD

The LAG and LEAD functions are useful for comparing values when the relative positions of rows can be known reliably. They work by specifying the count of rows which separate the target row from the current row. Because the functions provide access to more than one row of a table at the same time without a self-join, they can enhance processing speed. The LAG function provides access to a row at a given offset prior to the current position, and the LEAD function provides access to a row at a given offset after the current position.

The LAG and LEAD functions can be thought of as being related to, and a simplification of, the NTH_VALUE function. With LAG and LEAD, you can only retrieve values from a row at the specified physical offset. If this is insufficient, you can use NTH_VALUE, which enables you to retrieve values from a row based on what is called a logical offset or relative position. You can use the IGNORE NULLS option with the NTH_VALUE function to make it more useful, in the sense that you can specify conditions and filter out rows based on certain conditions. See Example 22–17, "NTH_VALUE", where rows with quantities less than eight are filtered out. This cannot be done with LAG or LEAD, as you would not know the offset to the row.

See "NTH_VALUE Function" on page 22-20 and *Oracle Database SQL Language Reference* for more information.

LAG/LEAD Syntax

These functions have the following syntax:

```
{LAG | LEAD} ( value_expr [, offset] [, default] ) [RESPECT NULLS|IGNORE NULLS]
   OVER ( [query_partition_clause] order_by_clause )
```

offset is an optional parameter and defaults to 1. *default* is an optional parameter and is the value returned if *offset* falls outside the bounds of the table or partition. When IGNORE NULLS is specified, the value returned will be from a row at a specified lag or lead offset after ignoring rows with NULLs.

Example 22–14 LAG/LEAD

This example illustrates a typical case of using LAG and LEAD:

```
SELECT time_id, TO_CHAR(SUM(amount_sold),'9,999,999') AS SALES,
  TO_CHAR(LAG(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LAG1,
  TO_CHAR(LEAD(SUM(amount_sold),1) OVER (ORDER BY time_id),'9,999,999') AS LEAD1
FROM sales
WHERE time_id>=TO_DATE('10-OCT-2000') AND time_id<=TO_DATE('14-OCT-2000')
GROUP BY time_id;

TIME_ID   SALES      LAG1       LEAD1
--------- ---------- ---------- ----------
10-OCT-00    238,479                23,183
11-OCT-00     23,183    238,479     24,616
12-OCT-00     24,616     23,183     76,516
13-OCT-00     76,516     24,616     29,795
14-OCT-00     29,795     76,516
```

See "Data Densification for Reporting" on page 22-37 for information showing how to use the LAG/LEAD functions for doing period-to-period comparison queries on sparse data.

Example 22–15 LAG/LEAD Using IGNORE NULLS

This example illustrates a typical case of using LAG and LEAD with the IGNORE NULLS option:

```
SELECT prod_id, channel_id, SUM(quantity_sold) quantity,
 CASE WHEN SUM(quantity_sold) < 5000 THEN SUM(amount_sold) ELSE NULL END amount,
 LAG(CASE WHEN SUM(quantity_sold) < 5000 THEN SUM(amount_sold) ELSE NULL END)
 IGNORE NULLS OVER (PARTITION BY prod_id ORDER BY channel_id) lag
FROM sales
WHERE prod_id IN (18,127,138)
GROUP BY prod_id, channel_id;

 PROD_ID  CHANNEL_ID  QUANTITY   AMOUNT          LAG
--------  ----------  --------   -----------     ----------
      18           2      2888    4420923.94
      18           3      5615                    4420923.94
      18           4      1088    1545729.81      4420923.94
     127           2      4508     274088.08
     127           3      9626                     274088.08
     127           4      1850     173682.67       274088.08
     138           2      1120      127390.3
     138           3      3878     393111.15        127390.3
     138           4       543      71203.21       393111.15

9 rows selected.
```

FIRST_VALUE, LAST_VALUE, and NTH_VALUE Functions

This section illustrates the FIRST_VALUE, LAST_VALUE, and NTH_VALUE functions.

FIRST_VALUE and LAST_VALUE Functions

The FIRST_VALUE and LAST_VALUE functions allow you to select the first and last rows from a window. These rows are especially valuable because they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, you might ask "How much was each day's sales compared to the first sales day (FIRST_VALUE) of the period?"

If the IGNORE NULLS option is used with FIRST_VALUE, it returns the first non-null value in the set, or NULL if all values are NULL. If IGNORE NULLS is used with LAST_VALUE, it returns the last non-null value in the set, or NULL if all values are NULL. The IGNORE NULLS option is particularly useful in populating an inventory table properly.

These functions have syntax as follows:

```
FIRST_VALUE|LAST_VALUE ( <expr> ) [RESPECT NULLS|IGNORE NULLS] OVER (analytic
clause );
```

Example 22–16 FIRST_VALUE

This example illustrates using the IGNORE NULLS option with FIRST_VALUE:

```
SELECT prod_id, channel_id, time_id,
  CASE WHEN MIN(amount_sold) > 9.5
    THEN MIN(amount_sold) ELSE NULL END amount_sold,
  FIRST_VALUE(CASE WHEN MIN(amount_sold) > 9.5
    THEN min(amount_sold) ELSE NULL END)
    IGNORE NULLS OVER (PARTITION BY prod_id
                       ORDER BY channel_id DESC, time_id
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND UNBOUNDED FOLLOWING) nv FROM sales
                       WHERE prod_id = 115 AND time_id BETWEEN '18-DEC-01'
                       AND '22-DEC-01' GROUP BY prod_id, channel_id, time_id
                       ORDER BY prod_id;

  PROD_ID CHANNEL_ID TIME_ID    AMOUNT_SOLD         NV
---------- ---------- --------- ----------- ----------
       115          4 18-DEC-01                   9.66
       115          4 19-DEC-01                   9.66
       115          4 20-DEC-01                   9.66
       115          4 22-DEC-01                   9.66
       115          3 18-DEC-01        9.66       9.66
       115          3 19-DEC-01        9.66       9.66
       115          3 20-DEC-01        9.66       9.66
       115          3 21-DEC-01        9.66       9.66
       115          3 22-DEC-01        9.66       9.66
       115          2 18-DEC-01        9.67       9.66
       115          2 19-DEC-01        9.67       9.66
       115          2 21-DEC-01        9.67       9.66
       115          2 22-DEC-01        9.67       9.66

13 rows selected.
```

NTH_VALUE Function

The NTH_VALUE function enables you to find column values from an arbitrary row in the window. This could be used when, for example, you want to retrieve the 5th highest closing price for a company's shares during a year.

The LAG and LEAD functions can be thought of as being related to, and a simplification of, the NTH_VALUE function. With LAG and LEAD, you can only retrieve values from a row at the specified physical offset. If this is insufficient, you can use NTH_VALUE,

which enables you to retrieve values from a row based on what is called a logical offset or relative position. You can use the IGNORE NULLS option with the NTH_VALUE, FIRST_VALUE, and LAST_VALUE functions to make it more powerful, in the sense that you can specify conditions and filter out rows based on certain conditions. See Example 22–17, "NTH_VALUE", where rows with quantities less than eight are filtered out. This cannot be done with LAG or LEAD, as you would not know the offset to the row.

See *Oracle Database SQL Language Reference* for more information.

This function has syntax as follows:

```
NTH_VALUE (<expr>, <n expr>) [FROM FIRST | FROM LAST]
[RESPECT NULLS | IGNORE NULLS] OVER (<window specification>)
```

- expr can be a column, constant, bind variable, or an expression involving them.
- n can be a column, constant, bind variable, or an expression involving them.
- RESPECT NULLS is the default NULL handling mechanism. It determines whether null values of expr are included in or eliminated from the calculation. The default is RESPECT NULLS.
- The FROM FIRST and FROM LAST options determine whether the offset n is from the first or last row. The default is FROM FIRST.
- IGNORE NULLS enables you to skip NULLs in measure values.

Example 22–17 NTH_VALUE

The following example returns the amount_sold value of the second channel_id in ascending order for each prod_id in the range between 10 and 20:

```
SELECT prod_id, channel_id, MIN(amount_sold),
  NTH_VALUE(MIN(amount_sold), 2) OVER (PARTITION BY prod_id ORDER BY channel_id
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) NV
FROM sales
WHERE prod_id BETWEEN 10 AND 20 GROUP BY prod_id, channel_id;

   PROD_ID CHANNEL_ID MIN(AMOUNT_SOLD)         NV
---------- ---------- ---------------- ----------
        13          2           907.34      906.2
        13          3            906.2      906.2
        13          4           842.21      906.2
        14          2          1015.94    1036.72
        14          3          1036.72    1036.72
        14          4           935.79    1036.72
        15          2           871.19     871.19
        15          3           871.19     871.19
        15          4           871.19     871.19
        16          2           266.84     266.84
        16          3           266.84     266.84
        16          4           266.84     266.84
        16          9            11.99     266.84
...
```

Advanced Aggregates for Analysis

This section illustrates the following advanced analytic aggregate functions:

- LISTAGG Function

- FIRST/LAST Functions
- Inverse Percentile
- Hypothetical Rank
- Linear Regression
- Statistical Aggregates
- User-Defined Aggregates

LISTAGG Function

The LISTAGG function orders data within each group based on the ORDER BY clause and then concatenates the values of the measure column. Its syntax is as follows:

```
LISTAGG (<expr> [, <delimiter>) WITHIN GROUP (ORDER BY <oby_expression_list>)
```

expr can be a column, constant, bind variable, or an expression involving them.

delimiter can be a column, constant, bind variable, or an expression involving them.

oby_expression_list can be a list of expressions with optional ordering options to sort in ascending or descending order (ASC or DESC), and to control the sort order of NULLs (NULLS FIRST or NULLS LAST). ASCENDING and NULLS LAST are the defaults.

LISTAGG as Aggregate

You can use the LISTAGG function as an aggregate.

Example 22–18 LISTAGG as Aggregate

The following example illustrates using LISTAGG as an aggregate.

```
SELECT prod_id, LISTAGG(cust_first_name||' '||cust_last_name, '; ')
  WITHIN GROUP (ORDER BY amount_sold DESC) cust_list
FROM sales, customers
WHERE sales.cust_id = customers.cust_id AND cust_gender = 'M'
  AND cust_credit_limit = 15000 AND prod_id BETWEEN 15 AND 18
  AND channel_id = 2 AND time_id > '01-JAN-01'
GROUP BY prod_id;

PROD_ID  CUST_LIST
-------  ------------------------------------------------
     15  Hope Haarper; Roxanne Crocker; ... Mason Murray
     16  Manvil Austin; Bud Pinkston; ... Helga Nickols
     17  Opal Aaron; Thacher Rudder; ... Roxanne Crocker
     18  Boyd Lin; Bud Pinkston; ... Erik Ready
```

The output has been modified for readability.

LISTAGG as Reporting Aggregate

You can use the LISTAGG function as a reporting aggregate.

Example 22–19 LISTAGG as Reporting Aggregate

This example illustrates using LISTAGG as a reporting aggregate:

```
connect sh/sh
set lines 120 pages 20000
```

```
column list format A40

SELECT time_id, prod_id, MIN(amount_sold), LISTAGG(min(amount_sold),';')
WITHIN GROUP (ORDER BY prod_id) OVER (PARTITION BY time_id) cust_list
FROM sales WHERE time_id > '20-DEC-01' AND prod_id BETWEEN 120 AND 125
GROUP BY prod_id, time_id;

TIME_ID   PROD_ID   MIN(AMOUNT_SOLD)  CUST_LIST
-------   -------   ----------------  -----------
21-DEC-01     120            51.36    51.36;10.81
21-DEC-01     121            10.81    51.36;10.81
22-DEC-01     120            51.36    51.36;10.81;20.23;56.12;17.79;15.67
22-DEC-01     121            10.81    51.36;10.81;20.23;56.12;17.79;15.67
22-DEC-01     122            20.23    51.36;10.81;20.23;56.12;17.79;15.67
22-DEC-01     123            56.12    51.36;10.81;20.23;56.12;17.79;15.67
22-DEC-01     124            17.79    51.36;10.81;20.23;56.12;17.79;15.67
22-DEC-01     125            15.67    51.36;10.81;20.23;56.12;17.79;15.67
...
```

FIRST/LAST Functions

The FIRST/LAST aggregate functions allow you to rank a data set and work with its top-ranked or bottom-ranked rows. After finding the top or bottom ranked rows, an aggregate function is applied to any desired column. That is, FIRST/LAST lets you rank on column A but return the result of an aggregate applied on the first-ranked or last-ranked rows of column B. This is valuable because it avoids the need for a self-join or subquery, thus improving performance. These functions' syntax begins with a regular aggregate function (MIN, MAX, SUM, AVG, COUNT, VARIANCE, STDDEV) that produces a single return value per group. To specify the ranking used, the FIRST/LAST functions add a new clause starting with the word KEEP.

These functions have the following syntax:

```
aggregate_function KEEP ( DENSE_RANK LAST ORDER BY
  expr [ DESC | ASC ] [NULLS { FIRST | LAST }]
  [, expr [ DESC | ASC ] [NULLS { FIRST | LAST }]]...)
[OVER query_partitioning_clause]
```

Note that the ORDER BY clause can take multiple expressions.

FIRST/LAST As Regular Aggregates

You can use the FIRST/LAST family of aggregates as regular aggregate functions.

Example 22–20 FIRST/LAST Example 1

The following query lets us compare minimum price and list price of our products. For each product subcategory within the Men's clothing category, it returns the following:

- List price of the product with the lowest minimum price
- Lowest minimum price
- List price of the product with the highest minimum price
- Highest minimum price

```
SELECT prod_subcategory, MIN(prod_list_price)
  KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price)) AS LP_OF_LO_MINP,
MIN(prod_min_price) AS LO_MINP,
MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
 AS LP_OF_HI_MINP,
```

```
MAX(prod_min_price) AS HI_MINP
FROM products WHERE prod_category='Electronics'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY LP_OF_LO_MINP LO_MINP  LP_OF_HI_MINP  HI_MINP
---------------- ------------- -------  -------------- ----------
Game Consoles          299.99  299.99          299.99     299.99
Home Audio             499.99  499.99          599.99     599.99
Y Box Accessories        7.99    7.99           20.99      20.99
Y Box Games              7.99    7.99           29.99      29.99
```

FIRST/LAST As Reporting Aggregates

You can also use the FIRST/LAST family of aggregates as reporting aggregate functions. An example is calculating which months had the greatest and least increase in head count throughout the year. The syntax for these functions is similar to the syntax for any other reporting aggregate.

Consider the example in Example 22–20 for FIRST/LAST. What if we wanted to find the list prices of individual products and compare them to the list prices of the products in their subcategory that had the highest and lowest minimum prices?

The following query lets us find that information for the Documentation subcategory by using FIRST/LAST as reporting aggregates.

Example 22–21 FIRST/LAST Example 2

```
SELECT prod_id, prod_list_price,
    MIN(prod_list_price) KEEP (DENSE_RANK FIRST ORDER BY (prod_min_price))
       OVER(PARTITION BY (prod_subcategory)) AS LP_OF_LO_MINP,
    MAX(prod_list_price) KEEP (DENSE_RANK LAST ORDER BY (prod_min_price))
       OVER(PARTITION BY (prod_subcategory)) AS LP_OF_HI_MINP
FROM products WHERE prod_subcategory = 'Documentation';

   PROD_ID PROD_LIST_PRICE LP_OF_LO_MINP LP_OF_HI_MINP
---------- --------------- ------------- -------------
        40           44.99         44.99         44.99
        41           44.99         44.99         44.99
        42           44.99         44.99         44.99
        43           44.99         44.99         44.99
        44           44.99         44.99         44.99
        45           44.99         44.99         44.99
```

Using the FIRST and LAST functions as reporting aggregates makes it easy to include the results in calculations such as "Salary as a percent of the highest salary."

Inverse Percentile

Using the CUME_DIST function, you can find the cumulative distribution (percentile) of a set of values. However, the inverse operation (finding what value computes to a certain percentile) is neither easy to do nor efficiently computed. To overcome this difficulty, the PERCENTILE_CONT and PERCENTILE_DISC functions were introduced. These can be used both as window reporting functions as well as normal aggregate functions.

These functions need a sort specification and a parameter that takes a percentile value between 0 and 1. The sort specification is handled by using an ORDER BY clause with one expression. When used as a normal aggregate function, it returns a single value for each ordered set.

PERCENTILE_CONT, which is a continuous function computed by interpolation, and PERCENTILE_DISC, which is a step function that assumes discrete values. Like other aggregates, PERCENTILE_CONT and PERCENTILE_DISC operate on a group of rows in a grouped query, but with the following differences:

- They require a parameter between 0 and 1 (inclusive). A parameter specified out of this range results in error. This parameter should be specified as an expression that evaluates to a constant.
- They require a sort specification. This sort specification is an ORDER BY clause with a single expression. Multiple expressions are not allowed.

Normal Aggregate Syntax

```
[PERCENTILE_CONT | PERCENTILE_DISC]( constant expression )
    WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
```

Inverse Percentile Example Basis

We use the following query to return the 17 rows of data used in the examples of this section:

```
SELECT cust_id, cust_credit_limit, CUME_DIST()
   OVER (ORDER BY cust_credit_limit) AS CUME_DIST
FROM customers WHERE cust_city='Marshal';

   CUST_ID CUST_CREDIT_LIMIT  CUME_DIST
---------- ----------------- ----------
     28344              1500 .173913043
      8962              1500 .173913043
     36651              1500 .173913043
     32497              1500 .173913043
     15192              3000 .347826087
    102077              3000 .347826087
    102343              3000 .347826087
      8270              3000 .347826087
     21380              5000  .52173913
     13808              5000  .52173913
    101784              5000  .52173913
     30420              5000  .52173913
     10346              7000 .652173913
     31112              7000 .652173913
     35266              7000 .652173913
      3424              9000 .739130435
    100977              9000 .739130435
    103066             10000 .782608696
     35225             11000 .956521739
     14459             11000 .956521739
     17268             11000 .956521739
    100421             11000 .956521739
     41496             15000          1
```

PERCENTILE_DISC(x) is computed by scanning up the CUME_DIST values in each group till you find the first one greater than or equal to x, where x is the specified percentile value. For the example query where PERCENTILE_DISC(0.5), the result is 5,000, as the following illustrates:

```
SELECT PERCENTILE_DISC(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc, PERCENTILE_CONT(0.5) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
```

```
 FROM customers WHERE cust_city='Marshal';

PERC_DISC   PERC_CONT
---------   ---------
     5000        5000
```

The result of PERCENTILE_CONT is computed by linear interpolation between rows after ordering them. To compute PERCENTILE_CONT(x), we first compute the row number = RN= (1+x*(n-1)), where *n* is the number of rows in the group and *x* is the specified percentile value. The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers CRN = CEIL(RN) and FRN = FLOOR(RN).

The final result is: PERCENTILE_CONT(X) = if (CRN = FRN = RN), then (value of expression from row at RN) else (CRN - RN) * (value of expression for row at FRN) + (RN -FRN) * (value of expression for row at CRN).

Consider the previous example query, where we compute PERCENTILE_CONT(0.5). Here *n* is 17. The row number RN = (1 + 0.5*(n-1))= 9 for both groups. Putting this into the formula, (FRN=CRN=9), we return the value from row 9 as the result.

Another example is, if you want to compute PERCENTILE_CONT(0.66). The computed row number RN=(1 + 0.66*(n-1))= (1 + 0.66*16)= 11.67. PERCENTILE_CONT(0.66) = (12-11.67)*(value of row 11)+(11.67-11)*(value of row 12). These results are:

```
SELECT PERCENTILE_DISC(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_disc, PERCENTILE_CONT(0.66) WITHIN GROUP
  (ORDER BY cust_credit_limit) AS perc_cont
FROM customers WHERE cust_city='Marshal';

 PERC_DISC  PERC_CONT
---------- ----------
      9000       8040
```

Inverse percentile aggregate functions can appear in the HAVING clause of a query like other existing aggregate functions.

As Reporting Aggregates

You can also use the aggregate functions PERCENTILE_CONT, PERCENTILE_DISC as reporting aggregate functions. When used as reporting aggregate functions, the syntax is similar to those of other reporting aggregates.

```
[PERCENTILE_CONT | PERCENTILE_DISC](constant expression)
WITHIN GROUP ( ORDER BY single order by expression
[ASC|DESC] [NULLS FIRST| NULLS LAST])
OVER ( [PARTITION BY value expression [,...]] )
```

This query performs the same computation (median credit limit for customers in this result set), but reports the result for every row in the result set, as shown in the following output:

```
SELECT cust_id, cust_credit_limit, PERCENTILE_DISC(0.5) WITHIN GROUP
     (ORDER BY cust_credit_limit) OVER () AS perc_disc,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY cust_credit_limit)
  OVER () AS perc_cont
FROM customers WHERE cust_city='Marshal';

   CUST_ID CUST_CREDIT_LIMIT  PERC_DISC  PERC_CONT
---------- ----------------- ---------- ----------
```

```
 28344            1500       5000       5000
  8962            1500       5000       5000
 36651            1500       5000       5000
 32497            1500       5000       5000
 15192            3000       5000       5000
102077            3000       5000       5000
102343            3000       5000       5000
  8270            3000       5000       5000
 21380            5000       5000       5000
 13808            5000       5000       5000
101784            5000       5000       5000
 30420            5000       5000       5000
 10346            7000       5000       5000
 31112            7000       5000       5000
 35266            7000       5000       5000
  3424            9000       5000       5000
100977            9000       5000       5000
103066           10000       5000       5000
 35225           11000       5000       5000
 14459           11000       5000       5000
 17268           11000       5000       5000
100421           11000       5000       5000
 41496           15000       5000       5000
```

Restrictions

For PERCENTILE_DISC, the expression in the ORDER BY clause can be of any data type that you can sort (numeric, string, date, and so on). However, the expression in the ORDER BY clause must be a numeric or datetime type (including intervals) because linear interpolation is used to evaluate PERCENTILE_CONT. If the expression is of type DATE, the interpolated result is rounded to the smallest unit for the type. For a DATE type, the interpolated value is rounded to the nearest second, for interval types to the nearest second (INTERVAL DAY TO SECOND) or to the month (INTERVAL YEAR TO MONTH).

Like other aggregates, the inverse percentile functions ignore NULLs in evaluating the result. For example, when you want to find the median value in a set, Oracle Database ignores the NULLs and finds the median among the non-null values. You can use the NULLS FIRST/NULLS LAST option in the ORDER BY clause, but they will be ignored as NULLs are ignored.

Hypothetical Rank

These functions provide functionality useful for what-if analysis. As an example, what would be the rank of a row, if the row was hypothetically inserted into a set of other rows?

This family of aggregates takes one or more arguments of a hypothetical row and an ordered group of rows, returning the RANK, DENSE_RANK, PERCENT_RANK or CUME_DIST of the row as if it was hypothetically inserted into the group.

```
[RANK | DENSE_RANK | PERCENT_RANK | CUME_DIST]( constant expression [, ...] )
WITHIN GROUP ( ORDER BY order by expression [ASC|DESC] [NULLS FIRST|NULLS LAST][,
...] )
```

Here, *constant expression* refers to an expression that evaluates to a constant, and there may be more than one such expressions that are passed as arguments to the function. The ORDER BY clause can contain one or more expressions that define the

sorting order on which the ranking will be based. ASC, DESC, NULLS FIRST, NULLS LAST options will be available for each expression in the ORDER BY.

Example 22–22 Hypothetical Rank and Distribution Example 1

Using the list price data from the products table used throughout this section, you can calculate the RANK, PERCENT_RANK and CUME_DIST for a hypothetical sweater with a price of $50 for how it fits within each of the sweater subcategories. The query and results are:

```
SELECT cust_city,
  RANK(6000) WITHIN GROUP (ORDER BY CUST_CREDIT_LIMIT DESC) AS HRANK,
  TO_CHAR(PERCENT_RANK(6000) WITHIN GROUP
  (ORDER BY cust_credit_limit),'9.999') AS HPERC_RANK,
  TO_CHAR(CUME_DIST (6000) WITHIN GROUP
  (ORDER BY cust_credit_limit),'9.999') AS HCUME_DIST
FROM customers
WHERE cust_city LIKE 'Fo%'
GROUP BY cust_city;

CUST_CITY                          HRANK HPERC_ HCUME_
------------------------------ ---------- ------ ------
Fondettes                              13   .455   .478
Fords Prairie                          18   .320   .346
Forest City                            47   .370   .378
Forest Heights                         38   .456   .464
Forestville                            58   .412   .418
Forrestcity                            51   .438   .444
Fort Klamath                           59   .356   .363
Fort William                           30   .500   .508
Foxborough                             52   .414   .420
```

Unlike the inverse percentile aggregates, the ORDER BY clause in the sort specification for hypothetical rank and distribution functions may take multiple expressions. The number of arguments and the expressions in the ORDER BY clause should be the same and the arguments must be constant expressions of the same or compatible type to the corresponding ORDER BY expression. The following is an example using two arguments in several hypothetical ranking functions.

Example 22–23 Hypothetical Rank and Distribution Example 2

```
SELECT prod_subcategory,
  RANK(10,8) WITHIN GROUP (ORDER BY prod_list_price DESC,prod_min_price)
  AS HRANK, TO_CHAR(PERCENT_RANK(10,8) WITHIN GROUP
  (ORDER BY prod_list_price, prod_min_price),'9.999') AS HPERC_RANK,
  TO_CHAR(CUME_DIST (10,8) WITHIN GROUP
  (ORDER BY prod_list_price, prod_min_price),'9.999') AS HCUME_DIST
FROM products WHERE prod_subcategory LIKE 'Recordable%'
GROUP BY prod_subcategory;

PROD_SUBCATEGORY          HRANK   HPERC_   HCUME_
--------------------      -----   ------   ------
Recordable CDs                4     .571     .625
Recordable DVD Discs          5     .200     .333
```

These functions can appear in the HAVING clause of a query just like other aggregate functions. They cannot be used as either reporting aggregate functions or windowing aggregate functions.

Linear Regression

The regression functions support the fitting of an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate functions or windowing or reporting functions.

The regression functions are as follows:

- REGR_COUNT Function
- REGR_AVGY and REGR_AVGX Functions
- REGR_SLOPE and REGR_INTERCEPT Functions
- REGR_R2 Function
- REGR_SXX, REGR_SYY, and REGR_SXY Functions

Oracle applies the function to the set of (e1, e2) pairs after eliminating all pairs for which either of e1 or e2 is null. e1 is interpreted as a value of the dependent variable (a "y value"), and e2 is interpreted as a value of the independent variable (an "x value"). Both expressions must be numbers.

The regression functions are all computed simultaneously during a single pass through the data. They are frequently combined with the COVAR_POP, COVAR_SAMP, and CORR functions.

REGR_COUNT Function

REGR_COUNT returns the number of non-null number pairs used to fit the regression line. If applied to an empty set (or if there are no (e1, e2) pairs where neither of e1 or e2 is null), the function returns 0.

REGR_AVGY and REGR_AVGX Functions

REGR_AVGY and REGR_AVGX compute the averages of the dependent variable and the independent variable of the regression line, respectively. REGR_AVGY computes the average of its first argument (e1) after eliminating (e1, e2) pairs where either of e1 or e2 is null. Similarly, REGR_AVGX computes the average of its second argument (e2) after null elimination. Both functions return NULL if applied to an empty set.

REGR_SLOPE and REGR_INTERCEPT Functions

The REGR_SLOPE function computes the slope of the regression line fitted to non-null (e1, e2) pairs.

The REGR_INTERCEPT function computes the y-intercept of the regression line. REGR_INTERCEPT returns NULL whenever slope or the regression averages are NULL.

REGR_R2 Function

The REGR_R2 function computes the coefficient of determination (usually called "R-squared" or "goodness of fit") for the regression line.

REGR_R2 returns values between 0 and 1 when the regression line is defined (slope of the line is not null), and it returns NULL otherwise. The closer the value is to 1, the better the regression line fits the data.

REGR_SXX, REGR_SYY, and REGR_SXY Functions

REGR_SXX, REGR_SYY and REGR_SXY functions are used in computing various diagnostic statistics for regression analysis. After eliminating (e1, e2) pairs where either of e1 or e2 is null, these functions make the following computations:

```
REGR_SXX:    REGR_COUNT(e1,e2) * VAR_POP(e2)
REGR_SYY:    REGR_COUNT(e1,e2) * VAR_POP(e1)
REGR_SXY:    REGR_COUNT(e1,e2) * COVAR_POP(e1, e2)
```

Linear Regression Statistics Examples

Some common diagnostic statistics that accompany linear regression analysis are given in Table 22–2, " Common Diagnostic Statistics and Their Expressions". Note that this release's new functions allow you to calculate all of these.

Table 22–2 Common Diagnostic Statistics and Their Expressions

Type of Statistic	Expression
Adjusted R2	`1-((1 - REGR_R2)*((REGR_COUNT-1)/(REGR_COUNT-2)))`
Standard error	`SQRT((REGR_SYY-(POWER(REGR_SXY,2)/REGR_SXX))/(REGR_COUNT-2))`
Total sum of squares	`REGR_SYY`
Regression sum of squares	`POWER(REGR_SXY,2) / REGR_SXX`
Residual sum of squares	`REGR_SYY - (POWER(REGR_SXY,2)/REGR_SXX)`
t statistic for slope	`REGR_SLOPE * SQRT(REGR_SXX)` / (Standard error)
t statistic for y-intercept	`REGR_INTERCEPT` / ((Standard error) * `SQRT((1/REGR_COUNT)+(POWER(REGR_AVGX,2)/REGR_SXX))`

Sample Linear Regression Calculation

In this example, we compute an ordinary-least-squares regression line that expresses the quantity sold of a product as a linear function of the product's list price. The calculations are grouped by sales channel. The values `SLOPE`, `INTCPT`, `RSQR` are slope, intercept, and coefficient of determination of the regression line, respectively. The (integer) value `COUNT` is the number of products in each channel for whom both quantity sold and list price data are available.

```
SELECT s.channel_id, REGR_SLOPE(s.quantity_sold, p.prod_list_price) SLOPE,
  REGR_INTERCEPT(s.quantity_sold, p.prod_list_price) INTCPT,
  REGR_R2(s.quantity_sold, p.prod_list_price) RSQR,
  REGR_COUNT(s.quantity_sold, p.prod_list_price) COUNT,
  REGR_AVGX(s.quantity_sold, p.prod_list_price) AVGLISTP,
  REGR_AVGY(s.quantity_sold, p.prod_list_price) AVGQSOLD
FROM sales s, products p WHERE s.prod_id=p.prod_id
  AND p.prod_category='Electronics' AND s.time_id=to_DATE('10-OCT-2000')
GROUP BY s.channel_id;

CHANNEL_ID      SLOPE     INTCPT       RSQR      COUNT   AVGLISTP   AVGQSOLD
---------- ---------- ---------- ---------- ---------- ---------- ----------
         2          0          1          1         39 466.656667          1
         3          0          1          1         60     459.99          1
         4          0          1          1         19 526.305789          1
```

Statistical Aggregates

Oracle provides a set of SQL statistical functions and a statistics package, `DBMS_STAT_FUNCS`. This section lists some of the new functions along with basic syntax.

See *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `DBMS_STAT_FUNCS` package and *Oracle Database SQL Language Reference* for syntax and semantics.

Descriptive Statistics

You can calculate the following descriptive statistics:

- Median of a Data Set

```
Median (expr) [OVER (query_partition_clause)]
```

- Mode of a Data Set

```
STATS_MODE (expr)
```

Hypothesis Testing - Parametric Tests

You can calculate the following descriptive statistics:

- One-Sample T-Test

```
STATS_T_TEST_ONE (expr1, expr2 (a constant) [, return_value])
```

- Paired-Samples T-Test

```
STATS_T_TEST_PAIRED (expr1, expr2 [, return_value])
```

- Independent-Samples T-Test. Pooled Variances

```
STATS_T_TEST_INDEP (expr1, expr2 [, return_value])
```

- Independent-Samples T-Test, Unpooled Variances

```
STATS_T_TEST_INDEPU (expr1, expr2 [, return_value])
```

- The F-Test

```
STATS_F_TEST (expr1, expr2 [, return_value])
```

- One-Way ANOVA

```
STATS_ONE_WAY_ANOVA (expr1, expr2 [, return_value])
```

Crosstab Statistics

You can calculate crosstab statistics using the following syntax:

```
STATS_CROSSTAB (expr1, expr2 [, return_value])
```

Can return any one of the following:

- Observed value of chi-squared
- Significance of observed chi-squared
- Degree of freedom for chi-squared
- Phi coefficient, Cramer's V statistic
- Contingency coefficient
- Cohen's Kappa

Hypothesis Testing - Non-Parametric Tests

You can calculate hypothesis statistics using the following syntax:

```
STATS_BINOMIAL_TEST (expr1, expr2, p [, return_value])
```

- Binomial Test/Wilcoxon Signed Ranks Test

```
STATS_WSR_TEST (expr1, expr2 [, return_value])
```

- Mann-Whitney Test

```
STATS_MW_TEST (expr1, expr2 [, return_value])
```

- Kolmogorov-Smirnov Test

```
STATS_KS_TEST (expr1, expr2 [, return_value])
```

Non-Parametric Correlation

You can calculate the following parametric statistics:

- Spearman's rho Coefficient

```
CORR_S (expr1, expr2 [, return_value])
```

- Kendall's tau-b Coefficient

```
CORR_K (expr1, expr2 [, return_value])
```

In addition to the functions, this release has a PL/SQL package, `DBMS_STAT_FUNCS`. It contains the descriptive statistical function `SUMMARY` along with functions to support distribution fitting. The `SUMMARY` function summarizes a numerical column of a table with a variety of descriptive statistics. The five distribution fitting functions support normal, uniform, Weibull, Poisson, and exponential distributions.

User-Defined Aggregates

Oracle offers a facility for creating your own functions, called user-defined aggregate functions. These functions are written in programming languages such as PL/SQL, Java, and C, and can be used as analytic functions or aggregates in materialized views. See *Oracle Database Data Cartridge Developer's Guide* for further information regarding syntax and restrictions.

The advantages of these functions are:

- Highly complex functions can be programmed using a fully procedural language.
- Higher scalability than other techniques when user-defined functions are programmed for parallel processing.
- Object data types can be processed.

As a simple example of a user-defined aggregate function, consider the skew statistic. This calculation measures if a data set has a lopsided distribution about its mean. It will tell you if one tail of the distribution is significantly larger than the other. If you created a user-defined aggregate called `udskew` and applied it to the credit limit data in the prior example, the SQL statement and results might look like this:

```
SELECT USERDEF_SKEW(cust_credit_limit) FROM customers
WHERE cust_city='Marshal';

USERDEF_SKEW
============
0.583891
```

Before building user-defined aggregate functions, you should consider if your needs can be met in regular SQL. Many complex calculations are possible directly in SQL, particularly by using the `CASE` expression.

Staying with regular SQL will enable simpler development, and many query operations are already well-parallelized in SQL. Even the earlier example, the skew statistic, can be created using standard, albeit lengthy, SQL.

Pivoting Operations

The data returned by business intelligence queries is often most usable if presented in a crosstabular format. The `pivot_clause` of the `SELECT` statement lets you write crosstabulation queries that rotate rows into columns, aggregating data in the process of the rotation. Pivoting is a key technique in data warehouses. In it, you transform multiple rows of input into fewer and generally wider rows in the data warehouse. When pivoting, an aggregation operator is applied for each item in the pivot column value list. The pivot column cannot contain an arbitrary expression. If you need to pivot on an expression, then you should alias the expression in a view before the `PIVOT` operation. The basic syntax is as follows:

```
SELECT ....
FROM <table-expr>
   PIVOT
     (
      aggregate-function(<column>)
      FOR <pivot-column> IN (<value1>, <value2>,..., <valuen>)
        ) AS <alias>
WHERE .....
```

See *Oracle Database SQL Language Reference* for `pivot_clause` syntax.

To illustrate the use of pivoting, create the following view as a basis for later examples:

```
CREATE VIEW sales_view AS
SELECT
 prod_name product, country_name country, channel_id channel,
 SUBSTR(calendar_quarter_desc, 6,2) quarter,
 SUM(amount_sold) amount_sold, SUM(quantity_sold) quantity_sold
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
  sales.prod_id = products.prod_id AND
  sales.cust_id = customers.cust_id AND
  customers.country_id = countries.country_id
GROUP BY prod_name, country_name, channel_id,
 SUBSTR(calendar_quarter_desc, 6, 2);
```

Example: Pivoting

The following statement illustrates a typical pivot on the `channel` column:

```
SELECT * FROM
  (SELECT product, channel, amount_sold
   FROM sales_view
   ) S PIVOT (SUM(amount_sold)
   FOR CHANNEL IN (3 AS DIRECT_SALES, 4 AS INTERNET_SALES,
                   5 AS CATALOG_SALES, 9 AS TELESALES))
ORDER BY product;

PRODUCT                  DIRECT_SALES  INTERNET_SALES  CATALOG_SALES  TELESALES
-----------------------  ------------  --------------  -------------  ---------
...
Internal 6X CD-ROM          229512.97        26249.55
Internal 8X CD-ROM          286291.49        42809.44
Keyboard Wrist Rest         200959.84        38695.36                  1522.73
```

```
...
```

Note that the output has created four new aliased columns, DIRECT_SALES, INTERNET_SALES, CATALOG_SALES, and TELESALES, one for each of the pivot values. The output is a sum. If no alias is provided, the column heading will be the values of the IN-list.

Pivoting on Multiple Columns

You can pivot on more than one column. The following statement illustrates a typical multiple column pivot:

```
SELECT *
FROM
     (SELECT product, channel, quarter, quantity_sold
      FROM sales_view
     ) PIVOT (SUM(quantity_sold)
                FOR (channel, quarter) IN
                  ((5, '02') AS CATALOG_Q2,
                   (4, '01') AS INTERNET_Q1,
                   (4, '04') AS INTERNET_Q4,
                   (2, '02') AS PARTNERS_Q2,
                   (9, '03') AS TELE_Q3
                  )
                );

PRODUCT              CATALOG_Q2  INTERNET_Q1  INTERNET_Q4  PARTNERS_Q2   TELE_Q3
-------              ----------  -----------  -----------  -----------   -------
...
Bounce                                  347          632          954
...
Smash Up Boxing                         129          280          560
...
Comic Book Heroes                        47          155          275
...
```

Note that this example specifies a multi-column IN-list with column headings designed to match the IN-list members.

Pivoting: Multiple Aggregates

You can pivot with multiple aggregates, as shown in the following example:

```
SELECT *
FROM
     (SELECT product, channel, amount_sold, quantity_sold
      FROM sales_view
     ) PIVOT (SUM(amount_sold) AS sums,
              SUM(quantity_sold) AS sumq
              FOR channel IN (5, 4, 2, 9)
               )
ORDER BY product;

PRODUCT               5_SUMS  5_SUMQ    4_SUMS   4_SUMQ      2_SUMS   2_SUMQ    9_SUMS   9_SUMQ
-------------         ------  ------    ------   ------      ------   ------    ------   ------
O/S Doc Set English                   142780.36     3081   381397.99     8044   6028.66      134
O/S Doc Set French                     55503.58     1192   132000.77     2782
...
```

Note that the query creates column headings by concatenating the pivot values (or alias) with the alias of the aggregate function, plus an underscore.

Distinguishing PIVOT-Generated Nulls from Nulls in Source Data

You can distinguish between null values that are generated from the use of PIVOT and those that exist in the source data. The following example illustrates nulls that PIVOT generates.

The following query returns rows with 5 columns, column prod_id, and pivot resulting columns Q1, Q1_COUNT_TOTAL, Q2, Q2_COUNT_TOTAL. For each unique value of prod_id, Q1_COUNT_TOTAL returns the total number of rows whose qtr value is Q1, that is, and Q2_COUNT_TOTAL returns the total number of rows whose qtr value is Q2.

Assume we have a table sales2 of the following structure:

```
PROD_ID   QTR  AMOUNT_SOLD
-------   ---  -----------
100        Q1           10
100        Q1           20
100        Q2         NULL
200        Q1           50

SELECT *
FROM sales2
        PIVOT
         ( SUM(amount_sold), COUNT(*) AS count_total
           FOR qtr IN ('Q1', 'Q2')
         );

PROD_ID    "Q1"    "Q1_COUNT_TOTAL"  "Q2"        "Q2_COUNT_TOTAL"
-------    ----    ----------------  ---------   ----------------
    100      20                   2   NULL <1>                  1
    200      50                   1   NULL <2>                  0
```

From the result, we know that for prod_id 100, there are 2 sales rows for quarter Q1, and 1 sales row for quarter Q2; for prod_id 200, there is 1 sales row for quarter Q1, and no sales row for quarter Q2.

So, in Q2_COUNT_TOTAL, you can identify that NULL<1> comes from a row in the original table whose measure is of null value, while NULL<2> is due to no row being present in the original table for prod_id 200 in quarter Q2.

Unpivoting Operations

An unpivot does not reverse a PIVOT operation. Instead, it rotates data from columns into rows. If you are working with pivoted data, an UNPIVOT operation cannot reverse any aggregations that have been made by PIVOT or any other means.

To illustrate unpivoting, first create a pivoted table that includes four columns, for quarters of the year:

```
CREATE TABLE pivotedTable AS
SELECT *
FROM (SELECT product, quarter, quantity_sold, amount_sold
      FROM sales_view)
    PIVOT
    (
       SUM(quantity_sold) AS sumq, SUM(amount_sold) AS suma
       FOR quarter IN ('01' AS Q1, '02' AS Q2, '03' AS Q3, '04' AS Q4));
```

The table's contents resemble the following:

```
SELECT *
FROM pivotedTable
ORDER BY product;

PRODUCT         Q1_SUMQ  Q1_SUMA  Q2_SUMQ  Q2_SUMA   Q3_SUMQ  Q3_SUMA   Q4_SUMQ    Q4_SUMA
--------------- -------  -------  -------  --------  -------  --------  -------    ---------
1.44MB External   6098   58301.33    5112   49001.56    6050   56974.3     5848     55341.28
128MB Memory      1963  110763.63    2361  132123.12    3069  170710.4     2832    157736.6
17" LCD           1492 1812786.94    1387 1672389.06    1591 1859987.66    1540   1844008.11
...
```

The following `UNPIVOT` operation rotates the quarter columns into rows. For each product, there will be four rows, one for each quarter.

```
SELECT product, DECODE(quarter, 'Q1_SUMQ', 'Q1', 'Q2_SUMQ', 'Q2', 'Q3_SUMQ', 'Q3',
   'Q4_SUMQ', 'Q4') AS quarter, quantity_sold
FROM  pivotedTable
   UNPIVOT INCLUDE NULLS
       (quantity_sold
        FOR quarter IN (Q1_SUMQ, Q2_SUMQ, Q3_SUMQ, Q4_SUMQ))
ORDER BY product, quarter;

PRODUCT                          QU    QUANTITY_SOLD
-------                          --    -------------
1.44MB External 3.5" Diskette    Q1             6098
1.44MB External 3.5" Diskette    Q2             5112
1.44MB External 3.5" Diskette    Q3             6050
1.44MB External 3.5" Diskette    Q4             5848
128MB Memory Card                Q1             1963
128MB Memory Card                Q2             2361
128MB Memory Card                Q3             3069
128MB Memory Card                Q4             2832
...
```

Note the use of `INCLUDE NULLS` in this example. You can also use `EXCLUDE NULLS`, which is the default setting.

In addition, you can also unpivot using two columns, as in the following:

```
SELECT product, quarter, quantity_sold, amount_sold
FROM  pivotedTable
   UNPIVOT INCLUDE NULLS
       (
        (quantity_sold, amount_sold)
        FOR quarter IN ((Q1_SUMQ, Q1_SUMA) AS 'Q1', (Q2_SUMQ, Q2_SUMA) AS 'Q2',
(Q3_SUMQ, Q3_SUMA) AS 'Q3', (Q4_SUMQ, Q4_SUMA) AS 'Q4'))
ORDER BY product, quarter;

PRODUCT                          QU   QUANTITY_SOLD   AMOUNT_SOLD
-------------------------------  --   -------------   ------------
1.44MB External 3.5" Diskette    Q1            6098       58301.33
1.44MB External 3.5" Diskette    Q2            5112       49001.56
1.44MB External 3.5" Diskette    Q3            6050       56974.3
1.44MB External 3.5" Diskette    Q4            5848       55341.28
128MB Memory Card                Q1            1963      110763.63
128MB Memory Card                Q2            2361      132123.12
128MB Memory Card                Q3            3069      170710.4
128MB Memory Card                Q4            2832      157736.6
```

Wildcard and Subquery Pivoting with XML Operations

If you want to use a wildcard argument or subquery in your pivoting columns, you can do so with PIVOT XML syntax. With PIVOT XML, the output of the operation is properly formatted XML.

The following example illustrates using the wildcard keyword, ANY. It outputs XML that includes all channel values in sales_view:

```
SELECT *
FROM
   (SELECT product, channel, quantity_sold
    FROM sales_view
    ) PIVOT XML(SUM(quantity_sold)
                FOR channel IN (ANY)
               );
```

Note that the keyword ANY is available in PIVOT operations only as part of an XML operation. This output includes data for cases where the channel exists in the data set. Also note that aggregation functions must specify a GROUP BY clause to return multiple values, yet the pivot_clause does not contain an explicit GROUP BY clause. Instead, the pivot_clause performs an implicit GROUP BY.

The following example illustrates using a subquery. It outputs XML that includes all channel values and the sales data corresponding to each channel:

```
SELECT *
FROM
     (SELECT product, channel, quantity_sold
      FROM sales_view
     ) PIVOT XML(SUM(quantity_sold)
                 FOR channel IN (SELECT DISTINCT channel_id FROM CHANNELS)
                );
```

The output densifies the data to include all possible channels for each product.

Data Densification for Reporting

Data is normally stored in sparse form. That is, if no value exists for a given combination of dimension values, no row exists in the fact table. However, you may want to view the data in dense form, with rows for all combination of dimension values displayed even when no fact data exist for them. For example, if a product did not sell during a particular time period, you may still want to see the product for that time period with zero sales value next to it. Moreover, time series calculations can be performed most easily when data is dense along the time dimension. This is because dense data will fill a consistent number of rows for each period, which in turn makes it simple to use the analytic windowing functions with physical offsets. Data densification is the process of converting sparse data into dense form.

To overcome the problem of sparsity, you can use a partitioned outer join to fill the gaps in a time series or any other dimension. Such a join extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. Oracle logically partitions the rows in your query based on the expression you specify in the PARTITION BY clause. The result of a partitioned outer join is a UNION of the outer joins of each of the partitions in the logically partitioned table with the table on the other side of the join.

Note that you can use this type of join to fill the gaps in any dimension, not just the time dimension. Most of the examples here focus on the time dimension because it is the dimension most frequently used as a basis for comparisons.

Partition Join Syntax

The syntax for partitioned outer join extends the ANSI SQL `JOIN` clause with the phrase `PARTITION BY` followed by an expression list. The expressions in the list specify the group to which the outer join is applied. The following are the two forms of syntax normally used for partitioned outer join:

```
SELECT .....
FROM table_reference
PARTITION BY (expr [, expr ]... )
RIGHT OUTER JOIN table_reference

SELECT .....
FROM table_reference
LEFT OUTER JOIN table_reference
PARTITION BY {expr [,expr ]...)
```

Note that `FULL OUTER JOIN` is not supported with a partitioned outer join.

Sample of Sparse Data

A typical situation with a sparse dimension is shown in the following example, which computes the weekly sales and year-to-date sales for the product Bounce for weeks 20-30 in 2000 and 2001:

```
SELECT SUBSTR(p.Prod_Name,1,15) Product_Name, t.Calendar_Year Year,
 t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
FROM Sales s, Times t, Products p
WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
 p.Prod_name IN ('Bounce') AND t.Calendar_Year IN (2000,2001) AND
 t.Calendar_Week_Number BETWEEN 20 AND 30
GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number;

PRODUCT_NAME          YEAR       WEEK      SALES
--------------- ---------- ---------- ----------
Bounce                2000         20        801
Bounce                2000         21    4062.24
Bounce                2000         22    2043.16
Bounce                2000         23    2731.14
Bounce                2000         24    4419.36
Bounce                2000         27    2297.29
Bounce                2000         28    1443.13
Bounce                2000         29    1927.38
Bounce                2000         30    1927.38
Bounce                2001         20    1483.3
Bounce                2001         21    4184.49
Bounce                2001         22    2609.19
Bounce                2001         23    1416.95
Bounce                2001         24    3149.62
Bounce                2001         25    2645.98
Bounce                2001         27    2125.12
Bounce                2001         29    2467.92
Bounce                2001         30    2620.17
```

In this example, we would expect 22 rows of data (11 weeks each from 2 years) if the data were dense. However, we get only 18 rows because weeks 25 and 26 are missing in 2000, and weeks 26 and 28 in 2001.

Filling Gaps in Data

We can take the sparse data of the preceding query and do a partitioned outer join with a dense set of time data. In the following query, we alias our original query as `v` and we select data from the `times` table, which we alias as `t`. Here we retrieve 22 rows because there are no gaps in the series. The four added rows each have 0 as their Sales value set to 0 by using the `NVL` function.

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) dense_sales
FROM
 (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name,
  t.Calendar_Year Year, t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
  FROM Sales s, Times t, Products p
  WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
   p.Prod_name IN ('Bounce') AND t.Calendar_Year IN (2000,2001) AND
   t.Calendar_Week_Number BETWEEN 20 AND 30
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
 (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
  FROM Times
  WHERE Calendar_Year IN (2000, 2001)
  AND Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;

PRODUCT_NAME          YEAR       WEEK DENSE_SALES
--------------- ---------- ---------- -----------
Bounce                2000         20         801
Bounce                2000         21     4062.24
Bounce                2000         22     2043.16
Bounce                2000         23     2731.14
Bounce                2000         24     4419.36
Bounce                2000         25           0
Bounce                2000         26           0
Bounce                2000         27     2297.29
Bounce                2000         28     1443.13
Bounce                2000         29     1927.38
Bounce                2000         30     1927.38
Bounce                2001         20      1483.3
Bounce                2001         21     4184.49
Bounce                2001         22     2609.19
Bounce                2001         23     1416.95
Bounce                2001         24     3149.62
Bounce                2001         25     2645.98
Bounce                2001         26           0
Bounce                2001         27     2125.12
Bounce                2001         28           0
Bounce                2001         29     2467.92
Bounce                2001         30     2620.17
```

Note that in this query, a `WHERE` condition was placed for weeks between 20 and 30 in the inline view for the time dimension. This was introduced to keep the result set small.

Filling Gaps in Two Dimensions

N-dimensional data is typically displayed as a dense 2-dimensional cross tab of (n - 2) page dimensions. This requires that all dimension values for the two dimensions

appearing in the cross tab be filled in. The following is another example where the partitioned outer join capability can be used for filling the gaps on two dimensions:

```
WITH v1 AS
 (SELECT p.prod_id, country_id, calendar_year,
    SUM(quantity_sold) units, SUM(amount_sold) sales
  FROM sales s, products p, customers c, times t
  WHERE s.prod_id in (147, 148) AND t.time_id = s.time_id AND
    c.cust_id = s.cust_id AND p.prod_id = s.prod_id
  GROUP BY p.prod_id, country_id, calendar_year),
v2 AS                                  --countries to use for densifications
  (SELECT DISTINCT country_id
  FROM customers
  WHERE country_id IN (52782, 52785, 52786, 52787, 52788)),
v3 AS                                   --years to use for densifications
  (SELECT DISTINCT calendar_year FROM times)
SELECT v4.prod_id, v4.country_id, v3.calendar_year, units, sales
FROM
  (SELECT prod_id, v2.country_id, calendar_year, units, sales
   FROM v1 PARTITION BY (prod_id)
   RIGHT OUTER JOIN v2                  --densifies on country
   ON (v1.country_id = v2.country_id)) v4
PARTITION BY (prod_id,country_id)
RIGHT OUTER JOIN v3                     --densifies on year
ON (v4.calendar_year = v3.calendar_year)
ORDER BY 1, 2, 3;
```

In this query, the `WITH` subquery factoring clause `v1` summarizes sales data at the product, country, and year level. This result is sparse but users may want to see all the country, year combinations for each product. To achieve this, we take each partition of `v1` based on product values and outer join it on the country dimension first. This will give us all values of country for each product. We then take that result and partition it on product and country values and then outer join it on time dimension. This will give us all time values for each product and country combination.

```
   PROD_ID COUNTRY_ID CALENDAR_YEAR      UNITS      SALES
---------- ---------- ------------- ---------- ----------
       147      52782          1998
       147      52782          1999         29     209.82
       147      52782          2000         71     594.36
       147      52782          2001        345    2754.42
       147      52782          2002
       147      52785          1998          1       7.99
       147      52785          1999
       147      52785          2000
       147      52785          2001
       147      52785          2002
       147      52786          1998          1       7.99
       147      52786          1999
       147      52786          2000          2      15.98
       147      52786          2001
       147      52786          2002
       147      52787          1998
       147      52787          1999
       147      52787          2000
       147      52787          2001
       147      52787          2002
       147      52788          1998
       147      52788          1999
       147      52788          2000          1       7.99
```

```
147      52788          2001
147      52788          2002
148      52782          1998        139    4046.67
148      52782          1999        228    5362.57
148      52782          2000        251    5629.47
148      52782          2001        308    7138.98
148      52782          2002
148      52785          1998
148      52785          1999
148      52785          2000
148      52785          2001
148      52785          2002
148      52786          1998
148      52786          1999
148      52786          2000
148      52786          2001
148      52786          2002
148      52787          1998
148      52787          1999
148      52787          2000
148      52787          2001
148      52787          2002
148      52788          1998          4     117.23
148      52788          1999
148      52788          2000
148      52788          2001
148      52788          2002
```

Filling Gaps in an Inventory Table

An inventory table typically tracks quantity of units available for various products. This table is sparse: it only stores a row for a product when there is an event. For a sales table, the event is a sale, and for the inventory table, the event is a change in quantity available for a product. For example, consider the following inventory table:

```
CREATE TABLE invent_table (
product VARCHAR2(10),
time_id DATE,
quant NUMBER);

INSERT INTO invent_table VALUES
 ('bottle', TO_DATE('01/04/01', 'DD/MM/YY'), 10);
INSERT INTO invent_table VALUES
 ('bottle', TO_DATE('06/04/01', 'DD/MM/YY'), 8);
INSERT INTO invent_table VALUES
 ('can', TO_DATE('01/04/01', 'DD/MM/YY'), 15);
INSERT INTO invent_table VALUES
 ('can', TO_DATE('04/04/01', 'DD/MM/YY'), 11);
```

The inventory table now has the following rows:

```
PRODUCT    TIME_ID   QUANT
---------- --------- -----
bottle     01-APR-01    10
bottle     06-APR-01     8
can        01-APR-01    15
can        04-APR-01    11
```

For reporting purposes, users may want to see this inventory data differently. For example, they may want to see all values of time for each product. This can be

accomplished using partitioned outer join. In addition, for the newly inserted rows of missing time periods, users may want to see the values for quantity of units column to be carried over from the most recent existing time period. The latter can be accomplished using analytic window function LAST_VALUE value. Here is the query and the desired output:

```
WITH v1 AS
 (SELECT time_id
  FROM times
  WHERE times.time_id BETWEEN
   TO_DATE('01/04/01', 'DD/MM/YY')
   AND TO_DATE('07/04/01', 'DD/MM/YY'))
SELECT product, time_id, quant quantity,
  LAST_VALUE(quant IGNORE NULLS)
    OVER (PARTITION BY product ORDER BY time_id)
    repeated_quantity
FROM
 (SELECT product, v1.time_id, quant
  FROM invent_table PARTITION BY (product)
  RIGHT OUTER JOIN v1
  ON (v1.time_id = invent_table.time_id))
ORDER BY 1, 2;
```

The inner query computes a partitioned outer join on time within each product. The inner query densifies the data on the time dimension (meaning the time dimension will now have a row for each day of the week). However, the measure column quantity will have nulls for the newly added rows (see the output in the column quantity in the following results.

The outer query uses the analytic function LAST_VALUE. Applying this function partitions the data by product and orders the data on the time dimension column (time_id). For each row, the function finds the last non-null value in the window due to the option IGNORE NULLS, which you can use with both LAST_VALUE and FIRST_VALUE. We see the desired output in the column repeated_quantity in the following output:

```
PRODUCT    TIME_ID   QUANTITY REPEATED_QUANTITY
---------- --------- -------- -----------------
bottle     01-APR-01       10                10
bottle     02-APR-01                         10
bottle     03-APR-01                         10
bottle     04-APR-01                         10
bottle     05-APR-01                         10
bottle     06-APR-01        8                 8
bottle     07-APR-01                          8
can        01-APR-01       15                15
can        02-APR-01                         15
can        03-APR-01                         15
can        04-APR-01       11                11
can        05-APR-01                         11
can        06-APR-01                         11
can        07-APR-01                         11
```

Computing Data Values to Fill Gaps

Examples in previous section illustrate how to use partitioned outer join to fill gaps in one or more dimensions. However, the result sets produced by partitioned outer join have null values for columns that are not included in the PARTITION BY list.

Typically, these are measure columns. Users can make use of analytic SQL functions to replace those null values with a non-null value.

For example, the following query computes monthly totals for products 64MB Memory card and DVD-R Discs (product IDs 122 and 136) for the year 2000. It uses partitioned outer join to densify data for all months. For the missing months, it then uses the analytic SQL function AVG to compute the sales and units to be the average of the months when the product was sold.

If working in SQL*Plus, the following two commands wraps the column headings for greater readability of results:

```
col computed_units  heading  'Computed|_units'
col computed_sales  heading  'Computed|_sales'

WITH V AS
 (SELECT substr(p.prod_name,1,12) prod_name, calendar_month_desc,
     SUM(quantity_sold) units, SUM(amount_sold) sales
   FROM sales s, products p, times t
   WHERE s.prod_id IN (122,136) AND calendar_year = 2000
     AND t.time_id = s.time_id
     AND p.prod_id = s.prod_id
   GROUP BY p.prod_name, calendar_month_desc)
SELECT v.prod_name, calendar_month_desc, units, sales,
  NVL(units, AVG(units) OVER (partition by v.prod_name)) computed_units,
  NVL(sales, AVG(sales) OVER (partition by v.prod_name)) computed_sales
FROM
  (SELECT DISTINCT calendar_month_desc
   FROM times
   WHERE calendar_year = 2000) t
   LEFT OUTER JOIN V
   PARTITION BY (prod_name)
   USING (calendar_month_desc);
                                                    computed   computed
PROD_NAME    CALENDAR      UNITS      SALES     _units     _sales
------------ -------- ---------- ---------- ---------- ----------
64MB Memory  2000-01         112    4129.72        112    4129.72
64MB Memory  2000-02         190       7049        190       7049
64MB Memory  2000-03          47    1724.98         47    1724.98
64MB Memory  2000-04          20      739.4         20      739.4
64MB Memory  2000-05          47    1738.24         47    1738.24
64MB Memory  2000-06          20      739.4         20      739.4
64MB Memory  2000-07                          72.6666667    2686.79
64MB Memory  2000-08                          72.6666667    2686.79
64MB Memory  2000-09                          72.6666667    2686.79
64MB Memory  2000-10                          72.6666667    2686.79
64MB Memory  2000-11                          72.6666667    2686.79
64MB Memory  2000-12                          72.6666667    2686.79
DVD-R Discs, 2000-01         167     3683.5        167     3683.5
DVD-R Discs, 2000-02         152    3362.24        152    3362.24
DVD-R Discs, 2000-03         188    4148.02        188    4148.02
DVD-R Discs, 2000-04         144    3170.09        144    3170.09
DVD-R Discs, 2000-05         189    4164.87        189    4164.87
DVD-R Discs, 2000-06         145    3192.21        145    3192.21
DVD-R Discs, 2000-07                             124.25    2737.71
DVD-R Discs, 2000-08                             124.25    2737.71
DVD-R Discs, 2000-09           1      18.91          1      18.91
DVD-R Discs, 2000-10                             124.25    2737.71
DVD-R Discs, 2000-11                             124.25    2737.71
DVD-R Discs, 2000-12           8     161.84          8     161.84
```

Time Series Calculations on Densified Data

Densification is not just for reporting purpose. It also enables certain types of calculations, especially, time series calculations. Time series calculations are easier when data is dense along the time dimension. Dense data has a consistent number of rows for each time periods which in turn make it simple to use analytic window functions with physical offsets.

To illustrate, let us first take the example on "Filling Gaps in Data" on page 22-39, and let's add an analytic function to that query. In the following enhanced version, we calculate weekly year-to-date sales alongside the weekly sales. The NULL values that the partitioned outer join inserts in making the time series dense are handled in the usual way: the SUM function treats them as 0's.

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) Current_sales,
 SUM(Sales)
   OVER (PARTITION BY Product_Name, t.year ORDER BY t.week) Cumulative_sales
FROM
 (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name, t.Calendar_Year Year,
   t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
  FROM Sales s, Times t, Products p
  WHERE s.Time_id = t.Time_id AND
   s.Prod_id = p.Prod_id AND p.Prod_name IN ('Bounce') AND
   t.Calendar_Year IN (2000,2001) AND
   t.Calendar_Week_Number BETWEEN 20 AND 30
  GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
(SELECT DISTINCT
 Calendar_Week_Number Week, Calendar_Year Year
 FROM Times
 WHERE Calendar_Year in (2000, 2001)
 AND Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;

PRODUCT_NAME          YEAR       WEEK CURRENT_SALES CUMULATIVE_SALES
--------------- ---------- ---------- ------------- ----------------
Bounce                2000         20           801              801
Bounce                2000         21       4062.24          4863.24
Bounce                2000         22       2043.16           6906.4
Bounce                2000         23       2731.14          9637.54
Bounce                2000         24       4419.36          14056.9
Bounce                2000         25             0          14056.9
Bounce                2000         26             0          14056.9
Bounce                2000         27       2297.29         16354.19
Bounce                2000         28       1443.13         17797.32
Bounce                2000         29       1927.38          19724.7
Bounce                2000         30       1927.38         21652.08
Bounce                2001         20        1483.3           1483.3
Bounce                2001         21       4184.49          5667.79
Bounce                2001         22       2609.19          8276.98
Bounce                2001         23       1416.95          9693.93
Bounce                2001         24       3149.62         12843.55
Bounce                2001         25       2645.98         15489.53
Bounce                2001         26             0         15489.53
Bounce                2001         27       2125.12         17614.65
Bounce                2001         28             0         17614.65
Bounce                2001         29       2467.92         20082.57
Bounce                2001         30       2620.17         22702.74
```

Period-to-Period Comparison for One Time Level: Example

How do we use this feature to compare values across time periods? Specifically, how do we calculate a year-over-year sales comparison at the week level? The following query returns on the same row, for each product, the year-to-date sales for each week of 2001 with that of 2000.

Note that in this example we start with a `WITH` clause. This improves readability of the query and lets us focus on the partitioned outer join. If working in SQL*Plus, the following command wraps the column headings for greater readability of results:

```
col Weekly_ytd_sales_prior_year  heading  'Weekly_ytd|_sales_|prior_year'

WITH v AS
  (SELECT SUBSTR(p.Prod_Name,1,6) Prod, t.Calendar_Year Year,
     t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
   FROM Sales s, Times t, Products p
   WHERE s.Time_id = t.Time_id AND
     s.Prod_id = p.Prod_id AND p.Prod_name in ('Y Box') AND
     t.Calendar_Year in (2000,2001) AND
     t.Calendar_Week_Number BETWEEN 30 AND 40
   GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number)
SELECT Prod , Year, Week, Sales,
  Weekly_ytd_sales, Weekly_ytd_sales_prior_year
FROM
  (SELECT Prod, Year, Week, Sales, Weekly_ytd_sales,
     LAG(Weekly_ytd_sales, 1) OVER
       (PARTITION BY Prod , Week ORDER BY Year) Weekly_ytd_sales_prior_year
   FROM
     (SELECT v.Prod  Prod , t.Year Year, t.Week Week,
        NVL(v.Sales,0) Sales, SUM(NVL(v.Sales,0)) OVER
          (PARTITION BY v.Prod , t.Year ORDER BY t.week) weekly_ytd_sales
      FROM v
      PARTITION BY (v.Prod )
      RIGHT OUTER JOIN
        (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
         FROM Times
         WHERE Calendar_Year IN (2000, 2001)) t
      ON (v.week = t.week AND v.Year = t.Year)
     ) dense_sales
  ) year_over_year_sales
WHERE Year = 2001 AND Week BETWEEN 30 AND 40
ORDER BY 1, 2, 3;
```

PROD	YEAR	WEEK	SALES	WEEKLY_YTD_SALES	Weekly_ytd _sales_ prior_year
Y Box	2001	30	7877.45	7877.45	0
Y Box	2001	31	13082.46	20959.91	1537.35
Y Box	2001	32	11569.02	32528.93	9531.57
Y Box	2001	33	38081.97	70610.9	39048.69
Y Box	2001	34	33109.65	103720.55	69100.79
Y Box	2001	35	0	103720.55	71265.35
Y Box	2001	36	4169.3	107889.85	81156.29
Y Box	2001	37	24616.85	132506.7	95433.09
Y Box	2001	38	37739.65	170246.35	107726.96
Y Box	2001	39	284.95	170531.3	118817.4
Y Box	2001	40	10868.44	181399.74	120969.69

In the `FROM` clause of the inline view `dense_sales`, we use a partitioned outer join of aggregate view `v` and time view `t` to fill gaps in the sales data along the time

dimension. The output of the partitioned outer join is then processed by the analytic function SUM ... OVER to compute the weekly year-to-date sales (the weekly_ytd_sales column). Thus, the view dense_sales computes the year-to-date sales data for each week, including those missing in the aggregate view s. The inline view year_over_year_sales then computes the year ago weekly year-to-date sales using the LAG function. The LAG function labeled weekly_ytd_sales_prior_year specifies a PARTITION BY clause that pairs rows for the same week of years 2000 and 2001 into a single partition. We then pass an offset of 1 to the LAG function to get the weekly year to date sales for the prior year.

The outermost query block selects data from year_over_year_sales with the condition yr = 2001, and thus the query returns, for each product, its weekly year-to-date sales in the specified weeks of years 2001 and 2000.

Period-to-Period Comparison for Multiple Time Levels: Example

While the prior example shows us a way to create comparisons for a single time level, it would be even more useful to handle multiple time levels in a single query. For example, we could compare sales versus the prior period at the year, quarter, month and day levels. How can we create a query which performs a year-over-year comparison of year-to-date sales for all levels of our time hierarchy?

We will take several steps to perform this task. The goal is a single query with comparisons at the day, week, month, quarter, and year level. The steps are as follows:

1. We will create a view called cube_prod_time, which holds a hierarchical cube of sales aggregated across times and products.
2. Then we will create a view of the time dimension to use as an edge of the cube. The time edge, which holds a complete set of dates, will be partitioned outer joined to the sparse data in the view cube_prod_time.
3. Finally, for maximum performance, we will create a materialized view, mv_prod_time, built using the same definition as cube_prod_time.

For more information regarding hierarchical cubes, see Chapter 21, "SQL for Aggregation in Data Warehouses". The materialized view is defined in Step 1 in the following section.

Step 1 Create the hierarchical cube view

The materialized view shown in the following may already exist in your system; if not, create it now. If you must generate it, note that we limit the query to just two products to keep processing time short:

```
CREATE OR REPLACE VIEW cube_prod_time AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3')
  END) Hierarchical_Time,
  calendar_year year, calendar_quarter_desc quarter,
  calendar_month_desc month, t.time_id day,
```

```
  prod_category cat, prod_subcategory subcat, p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
  SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
  p.prod_name IN ('Bounce', 'Y Box') AND s.prod_id = p.prod_id
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```

Because this view is limited to two products, it returns just over 2200 rows. Note that the column `Hierarchical_Time` contains string representations of time from all levels of the time hierarchy. The `CASE` expression used for the `Hierarchical_Time` column appends a marker (_0, _1, ...) to each date string to denote the time level of the value. A _0 represents the year level, _1 is quarters, _2 is months, and _3 is day. Note that the `GROUP BY` clause is a concatenated `ROLLUP` which specifies the rollup hierarchy for the time and product dimensions. The `GROUP BY` clause is what determines the hierarchical cube contents.

Step 2 Create the view edge_time, which is a complete set of date values

`edge_time` is the source for filling time gaps in the hierarchical cube using a partitioned outer join. The column `Hierarchical_Time` in `edge_time` will be used in a partitioned join with the `Hierarchical_Time` column in the view `cube_prod_time`. The following statement defines `edge_time`:

```
CREATE OR REPLACE VIEW edge_time AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(time_id) || '_3')
   END) Hierarchical_Time,
   calendar_year yr, calendar_quarter_number qtr_num,
   calendar_quarter_desc qtr, calendar_month_number mon_num,
   calendar_month_desc mon, time_id - TRUNC(time_id, 'YEAR') + 1 day_num,
   time_id day,
GROUPING_ID(calendar_year, calendar_quarter_desc,
  calendar_month_desc, time_id) gid_t
FROM TIMES
GROUP BY ROLLUP
 (calendar_year, (calendar_quarter_desc, calendar_quarter_number),
 (calendar_month_desc, calendar_month_number), time_id);
```

Step 3 Create the materialized view mv_prod_time to support faster performance

The materialized view definition is a duplicate of the view `cube_prod_time` defined earlier. Because it is a duplicate query, references to `cube_prod_time` will be rewritten to use the `mv_prod_time` materialized view. The following materialized

may already exist in your system; if not, create it now. If you must generate it, note that we limit the query to just two products to keep processing time short.

```
CREATE MATERIALIZED VIEW mv_prod_time
REFRESH COMPLETE ON DEMAND AS
SELECT
  (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
       THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
       THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
       THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3')
  END) Hierarchical_Time,
  calendar_year year, calendar_quarter_desc quarter,
  calendar_month_desc month, t.time_id day,
  prod_category cat, prod_subcategory subcat, p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
  SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
  p.prod_name IN ('Bounce', 'Y Box') AND s.prod_id = p.prod_id
GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```

Step 4 Create the comparison query

We have now set the stage for our comparison query. We can obtain period-to-period comparison calculations at all time levels. It requires applying analytic functions to a hierarchical cube with dense data along the time dimension.

Some of the calculations we can achieve for each time level are:

- Sum of sales for prior period at all levels of time.
- Variance in sales over prior period.
- Sum of sales in the same period a year ago at all levels of time.
- Variance in sales over the same period last year.

The following example performs all four of these calculations. It uses a partitioned outer join of the views `cube_prod_time` and `edge_time` to create an inline view of dense data called `dense_cube_prod_time`. The query then uses the `LAG` function in the same way as the prior single-level example. The outer `WHERE` clause specifies time at three levels: the days of August 2001, the entire month, and the entire third quarter of 2001. Note that the last two rows of the results contain the month level and quarter level aggregations.

Note that to make the results easier to read if you are using SQL*Plus, the column headings should be adjusted with the following commands. The commands will fold the column headings to reduce line length:

```
col sales_prior_period heading 'sales_prior|_period'
```

```
col variance_prior_period heading 'variance|_prior|_period'
col sales_same_period_prior_year heading 'sales_same|_period_prior|_year'
col variance_same_period_p_year heading 'variance|_same_period|_prior_year'
```

Here is the query comparing current sales to prior and year ago sales:

```
SELECT SUBSTR(prod,1,4) prod, SUBSTR(Hierarchical_Time,1,12) ht,
  sales, sales_prior_period,
  sales - sales_prior_period variance_prior_period,
  sales_same_period_prior_year,
  sales - sales_same_period_prior_year variance_same_period_p_year
FROM
 (SELECT cat, subcat, prod, gid_p, gid_t,
    Hierarchical_Time, yr, qtr, mon, day, sales,
    LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
      gid_t ORDER BY yr, qtr, mon, day)
      sales_prior_period,
    LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
      gid_t, qtr_num, mon_num, day_num ORDER BY yr)
      sales_same_period_prior_year
  FROM
   (SELECT c.gid, c.cat, c.subcat, c.prod, c.gid_p,
      t.gid_t, t.yr, t.qtr, t.qtr_num, t.mon, t.mon_num,
      t.day, t.day_num, t.Hierarchical_Time, NVL(s_sold,0) sales
    FROM cube_prod_time c
    PARTITION BY (gid_p, cat, subcat, prod)
    RIGHT OUTER JOIN edge_time t
    ON ( c.gid_t = t.gid_t AND
      c.Hierarchical_Time = t.Hierarchical_Time)
   ) dense_cube_prod_time
 )                        --side by side current and prior year sales
WHERE prod IN (139) AND gid_p=0 AND    --1 product and product level data
  ( (mon IN ('2001-08' ) AND gid_t IN (0, 1)) OR --day and month data
  (qtr IN ('2001-03' ) AND gid_t IN (3)))        --quarter level data
ORDER BY day;
```

PROD	HT	SALES	sales_prior _period	variance _prior _period	sales_same _period_prior _year	variance _same_period _prior_year
139	01-AUG-01_3	0	0	0	0	0
139	02-AUG-01_3	1347.53	0	1347.53	0	1347.53
139	03-AUG-01_3	0	1347.53	-1347.53	42.36	-42.36
139	04-AUG-01_3	57.83	0	57.83	995.75	-937.92
139	05-AUG-01_3	0	57.83	-57.83	0	0
139	06-AUG-01_3	0	0	0	0	0
139	07-AUG-01_3	134.81	0	134.81	880.27	-745.46
139	08-AUG-01_3	1289.89	134.81	1155.08	0	1289.89
139	09-AUG-01_3	0	1289.89	-1289.89	0	0
139	10-AUG-01_3	0	0	0	0	0
139	11-AUG-01_3	0	0	0	0	0
139	12-AUG-01_3	0	0	0	0	0
139	13-AUG-01_3	0	0	0	0	0
139	14-AUG-01_3	0	0	0	0	0
139	15-AUG-01_3	38.49	0	38.49	1104.55	-1066.06
139	16-AUG-01_3	0	38.49	-38.49	0	0
139	17-AUG-01_3	77.17	0	77.17	1052.03	-974.86
139	18-AUG-01_3	2467.54	77.17	2390.37	0	2467.54
139	19-AUG-01_3	0	2467.54	-2467.54	127.08	-127.08
139	20-AUG-01_3	0	0	0	0	0
139	21-AUG-01_3	0	0	0	0	0

```
139  22-AUG-01_3          0          0          0          0          0
139  23-AUG-01_3    1371.43          0    1371.43          0    1371.43
139  24-AUG-01_3     153.96    1371.43   -1217.47     2091.3   -1937.34
139  25-AUG-01_3          0     153.96    -153.96          0          0
139  26-AUG-01_3          0          0          0          0          0
139  27-AUG-01_3    1235.48          0    1235.48          0    1235.48
139  28-AUG-01_3      173.3    1235.48   -1062.18    2075.64   -1902.34
139  29-AUG-01_3          0      173.3     -173.3          0          0
139  30-AUG-01_3          0          0          0          0          0
139  31-AUG-01_3          0          0          0          0          0
139  2001-08_2      8347.43    7213.21    1134.22    8368.98     -21.55
139  2001-03_1      24356.8   28862.14   -4505.34   24168.99     187.81
```

The first LAG function (sales_prior_period) partitions the data on gid_p, cat, subcat, prod, gid_t and orders the rows on all the time dimension columns. It gets the sales value of the prior period by passing an offset of 1. The second LAG function (sales_same_period_prior_year) partitions the data on additional columns qtr_num, mon_num, and day_num and orders it on yr so that, with an offset of 1, it can compute the year ago sales for the same period. The outermost SELECT clause computes the variances.

Creating a Custom Member in a Dimension: Example

In many analytical SQL tasks, it is helpful to define custom members in a dimension. For instance, you might define a specialized time period for analyses. You can use a partitioned outer join to temporarily add a member to a dimension. Note that the new SQL MODEL clause is suitable for creating more complex scenarios involving new members in dimensions. See Chapter 23, "SQL for Modeling" for more information on this topic.

As an example of a task, what if we want to define a new member for our time dimension? We want to create a 13th member of the Month level in our time dimension. This 13th month is defined as the summation of the sales for each product in the first month of each quarter of year 2001.

The solution has two steps. Note that we will build this solution using the views and tables created in the prior example. Two steps are required. First, create a view with the new member added to the appropriate dimension. The view uses a UNION ALL operation to add the new member. To query using the custom member, use a CASE expression and a partitioned outer join.

Our new member for the time dimension is created with the following view:

```
CREATE OR REPLACE VIEW time_c AS
(SELECT * FROM edge_time
UNION ALL
SELECT '2001-13_2', 2001, 5, '2001-05', 13, '2001-13', null, null,
8 -- <gid_of_mon>
FROM DUAL);
```

In this statement, the view time_c is defined by performing a UNION ALL of the edge_time view (defined in the prior example) and the user-defined 13th month. The gid_t value of 8 was chosen to differentiate the custom member from the standard members. The UNION ALL specifies the attributes for a 13th month member by doing a SELECT from the DUAL table. Note that the grouping id, column gid_t, is set to 8, and the quarter number is set to 5.

Then, the second step is to use an inline view of the query to perform a partitioned outer join of cube_prod_time with time_c. This step creates sales data for the 13th

month at each level of product aggregation. In the main query, the analytic function SUM is used with a CASE expression to compute the 13th month, which is defined as the summation of the first month's sales of each quarter.

```
SELECT * FROM
  (SELECT SUBSTR(cat,1,12) cat, SUBSTR(subcat,1,12) subcat,
     prod, mon, mon_num,
     SUM(CASE WHEN mon_num IN (1, 4, 7, 10)
           THEN s_sold
           ELSE NULL
         END)
       OVER (PARTITION BY gid_p, prod, subcat, cat, yr) sales_month_13
   FROM
     (SELECT c.gid, c.prod, c.subcat, c.cat, gid_p,
        t.gid_t, t.day, t.mon, t.mon_num,
        t.qtr, t.yr, NVL(s_sold,0) s_sold
      FROM cube_prod_time c
      PARTITION BY (gid_p, prod, subcat, cat)
      RIGHT OUTER JOIN time_c t
      ON (c.gid_t = t.gid_t AND
        c.Hierarchical_Time = t.Hierarchical_Time)
     )
  )
WHERE mon_num=13;

CAT          SUBCAT             PROD MON         MON_NUM SALES_MONTH_13
------------ ------------ ---------- -------- ---------- --------------
Electronics  Game Console         16 2001-13          13      762334.34
Electronics  Y Box Games         139 2001-13          13       75650.22
Electronics  Game Console             2001-13          13      762334.34
Electronics  Y Box Games              2001-13          13       75650.22
Electronics                           2001-13          13      837984.56
                                      2001-13          13      837984.56
```

The SUM function uses a CASE to limit the data to months 1, 4, 7, and 10 within each year. Due to the tiny data set, with just 2 products, the rollup values of the results are necessarily repetitions of lower level aggregations. For more realistic set of rollup values, you can include more products from the Game Console and Y Box Games subcategories in the underlying materialized view.

Miscellaneous Analysis and Reporting Capabilities

This section illustrates the following additional analytic capabilities:

- WIDTH_BUCKET Function
- Linear Algebra
- CASE Expressions
- Frequent Itemsets

WIDTH_BUCKET Function

For a given expression, the WIDTH_BUCKET function returns the bucket number that the result of this expression will be assigned after it is evaluated. You can generate equiwidth histograms with this function. Equiwidth histograms divide data sets into buckets whose interval size (highest value to lowest value) is equal. The number of rows held by each bucket will vary. A related function, NTILE, creates equiheight buckets.

Equiwidth histograms can be generated only for numeric, date or datetime types. So the first three parameters should be all numeric expressions or all date expressions. Other types of expressions are not allowed. If the first parameter is NULL, the result is NULL. If the second or the third parameter is NULL, an error message is returned, as a NULL value cannot denote any end point (or any point) for a range in a date or numeric value dimension. The last parameter (number of buckets) should be a numeric expression that evaluates to a positive integer value; 0, NULL, or a negative value will result in an error.

Buckets are numbered from 0 to (n+1). Bucket 0 holds the count of values less than the minimum. Bucket(n+1) holds the count of values greater than or equal to the maximum specified value.

WIDTH_BUCKET Syntax

The WIDTH_BUCKET takes four expressions as parameters. The first parameter is the expression that the equiwidth histogram is for. The second and third parameters are expressions that denote the end points of the acceptable range for the first parameter. The fourth parameter denotes the number of buckets.

```
WIDTH_BUCKET(expression, minval expression, maxval expression, num buckets)
```

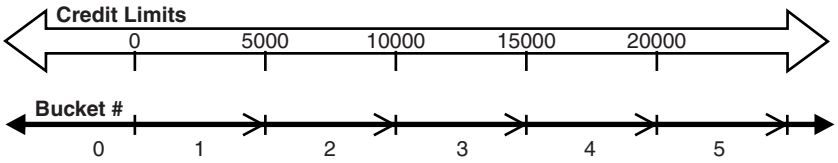
Consider the following data from table customers, that shows the credit limits of 17 customers. This data is gathered in the query shown in Example 22–24 on page 22-53.

```
CUST_ID    CUST_CREDIT_LIMIT
---------  -----------------
    10346               7000
    35266               7000
    41496              15000
    35225              11000
     3424               9000
    28344               1500
    31112               7000
     8962               1500
    15192               3000
    21380               5000
    36651               1500
    30420               5000
     8270               3000
    17268              11000
    14459              11000
    13808               5000
    32497               1500
   100977               9000
   102077               3000
   103066              10000
   101784               5000
   100421              11000
   102343               3000
```

In the table customers, the column cust_credit_limit contains values between 1500 and 15000, and we can assign the values to four equiwidth buckets, numbered from 1 to 4, by using WIDTH_BUCKET (cust_credit_limit, 0, 20000, 4). Ideally each bucket is a closed-open interval of the real number line, for example, bucket number 2 is assigned to scores between 5000.0000 and 9999.9999..., sometimes denoted [5000, 10000) to indicate that 5,000 is included in the interval and 10,000 is excluded. To accommodate values outside the range [0, 20,000), values less than 0 are assigned to a designated underflow bucket which is numbered 0, and values greater

than or equal to 20,000 are assigned to a designated overflow bucket which is numbered 5 (num buckets + 1 in general). See Figure 22–3 for a graphical illustration of how the buckets are assigned.

Figure 22–3 Bucket Assignments



You can specify the bounds in the reverse order, for example, `WIDTH_BUCKET (cust_credit_limit, 20000, 0, 4)`. When the bounds are reversed, the buckets will be open-closed intervals. In this example, bucket number 1 is `(15000, 20000]`, bucket number 2 is `(10000, 15000]`, and bucket number 4, is `(0, 5000]`. The overflow bucket will be numbered 0 `(20000, +infinity)`, and the underflow bucket will be numbered 5 `(-infinity, 0]`.

It is an error if the bucket count parameter is 0 or negative.

Example 22–24 WIDTH_BUCKET

The following query shows the bucket numbers for the credit limits in the customers table for both cases where the boundaries are specified in regular or reverse order. We use a range of 0 to 20,000.

```
SELECT cust_id, cust_credit_limit,
 WIDTH_BUCKET(cust_credit_limit,0,20000,4) AS WIDTH_BUCKET_UP,
 WIDTH_BUCKET(cust_credit_limit,20000, 0, 4) AS WIDTH_BUCKET_DOWN
FROM customers WHERE cust_city = 'Marshal';

   CUST_ID CUST_CREDIT_LIMIT WIDTH_BUCKET_UP WIDTH_BUCKET_DOWN
---------- ----------------- --------------- -----------------
     10346              7000               2                 3
     35266              7000               2                 3
     41496             15000               4                 2
     35225             11000               3                 2
      3424              9000               2                 3
     28344              1500               1                 4
     31112              7000               2                 3
      8962              1500               1                 4
     15192              3000               1                 4
     21380              5000               2                 4
     36651              1500               1                 4
     30420              5000               2                 4
      8270              3000               1                 4
     17268             11000               3                 2
     14459             11000               3                 2
     13808              5000               2                 4
     32497              1500               1                 4
    100977              9000               2                 3
    102077              3000               1                 4
    103066             10000               3                 3
    101784              5000               2                 4
    100421             11000               3                 2
    102343              3000               1                 4
```

Linear Algebra

Linear algebra is a branch of mathematics with a wide range of practical applications. Many areas have tasks that can be expressed using linear algebra, and here are some examples from several fields: statistics (multiple linear regression and principle components analysis), data mining (clustering and classification), bioinformatics (analysis of microarray data), operations research (supply chain and other optimization problems), econometrics (analysis of consumer demand data), and finance (asset allocation problems). Various libraries for linear algebra are freely available for anyone to use. Oracle's `UTL_NLA` package exposes matrix PL/SQL data types and wrapper PL/SQL subprograms for two of the most popular and robust of these libraries, BLAS and LAPACK.

Linear algebra depends on matrix manipulation. Performing matrix manipulation in PL/SQL in the past required inventing a matrix representation based on PL/SQL's native data types and then writing matrix manipulation routines from scratch. This required substantial programming effort and the performance of the resulting implementation was limited. If developers chose to send data to external packages for processing rather than create their own routines, data transfer back and forth could be time consuming. Using the `UTL_NLA` package lets data stay within Oracle, removes the programming effort, and delivers a fast implementation.

Example 22–25 Linear Algebra

Here is an example of how Oracle's linear algebra support could be used for business analysis. It invokes a multiple linear regression application built using the `UTL_NLA` package. The multiple regression application is implemented in an object called `OLS_Regression`. Note that sample files for the OLS Regression object can be found in `$ORACLE_HOME/plsql/demo`.

Consider the scenario of a retailer analyzing the effectiveness of its marketing program. Each of its stores allocates its marketing budget over the following possible programs: media advertisements (`media`), promotions (`promo`), discount coupons (`disct`), and direct mailers (`dmail`). The regression analysis builds a linear relationship between the amount of sales that an average store has in a given year (`sales`) and the spending on the four components of the marketing program. Suppose that the marketing data is stored in the following table:

```
sales_marketing_data (
  /* Store information*/
  store_no   NUMBER,
  year       NUMBER,
  /* Sales revenue (in dollars)*/
  sales      NUMBER,   /* sales amount*/
  /* Marketing expenses (in dollars)*/
  media      NUMBER,   /*media advertisements*/
  promo      NUMBER,   /*promotions*/
  disct      NUMBER,   /*dicount coupons*/
  dmail      NUMBER,   /*direct mailers*/
```

Then you can build the following sales-marketing linear model using coefficients:

```
Sales Revenue = a  + b Media Advisements
                   + c Promotions
                   + d Discount Coupons
                   + e Direct Mailer
```

This model can be implemented as the following view, which refers to the OLS regression object:

```
CREATE OR REPLACE VIEW sales_marketing_model (year, ols)
   AS SELECT year,
        OLS_Regression(
        /* mean_y => */
        AVG(sales),
        /* variance_y => */
        var_pop(sales),
        /* MV mean vector => */
        UTL_NLA_ARRAY_DBL (AVG(media),AVG(promo),
                           AVG(disct),AVG(dmail)),
        /* VCM variance covariance matrix => */
        UTL_NLA_ARRAY_DBL (var_pop(media),covar_pop(media,promo),
                           covar_pop(media,disct),covar_pop(media,dmail),
                           var_pop(promo),covar_pop(promo,disct),
                           covar_pop(promo,dmail),var_pop(disct),
                           covar_pop(disct,dmail),var_pop(dmail)),
        /* CV covariance vector => */
  UTL_NLA_ARRAY_DBL (covar_pop(sales,media),covar_pop(sales,promo),
                           covar_pop(sales,disct),covar_pop(sales,dmail)))
 FROM sales_marketing_data
 GROUP BY year;
```

Using this view, a marketing program manager can perform an analysis such as "Is this sales-marketing model reasonable for year 2004 data? That is, is the multiple-correlation greater than some acceptable value, say, 0.9?" The SQL for such a query might be as follows:

```
SELECT model.ols.getCorrelation(1)
       AS "Applicability of Linear Model"
FROM sales_marketing_model model
WHERE year = 2004;
```

You could also solve questions such as "What is the expected base-line sales revenue of a store without any marketing programs in 2003?" or "Which component of the marketing program was the most effective in 2004? That is, a dollar increase in which program produced the greatest expected increase in sales?"

See *Oracle Database PL/SQL Packages and Types Reference* for further information regarding the use of the `UTL_NLA` package and linear algebra.

CASE Expressions

Oracle now supports simple and searched `CASE` statements. `CASE` statements are similar in purpose to the `DECODE` statement, but they offer more flexibility and logical power. They are also easier to read than traditional `DECODE` statements, and offer better performance as well. They are commonly used when breaking categories into buckets like age (for example, 20-29, 30-39, and so on).

The syntax for simple `CASE` statements is:

```
CASE expr WHEN comparison_expr THEN return_expr
 [, WHEN comparison_expr THEN return_expr]... [ELSE else_expr] END
```

Simple `CASE` expressions test if the `expr` value equals the `comparison_expr`.

The syntax for searched `CASE` statements is:

```
CASE WHEN condition THEN return_expr [, WHEN condition THEN return_expr]
   ... ELSE else_expr] END
```

You can use any kind of condition in a searched CASE expression, not just an equality test.

You can specify only 65,535 arguments and each WHEN ... THEN pair counts as two arguments. To avoid exceeding this limit, you can nest CASE expressions so that the return_expr itself is a CASE expression.

Example 22–26 CASE

Suppose you wanted to find the average salary of all employees in the company. If an employee's salary is less than $2000, you want the query to use $2000 instead. Without a CASE statement, you might choose to write this query as follows:

```
SELECT AVG(foo(e.salary)) FROM employees e;
```

Note that this runs against the hr sample schema. In this, foo is a function that returns its input if the input is greater than 2000, and returns 2000 otherwise. The query has performance implications because it needs to invoke a function for each row. Writing custom functions can also add to the development load.

Using CASE expressions in the database without PL/SQL, this query can be rewritten as:

```
SELECT AVG(CASE when e.salary > 2000 THEN e.salary ELSE 2000 end)
  AS avg_sal_2k_floor
FROM employees e;
```

Using a CASE expression lets you avoid developing custom functions and can also perform faster.

Example 22–27 CASE for Aggregating Independent Subsets

Using CASE inside aggregate functions is a convenient way to perform aggregates on multiple subsets of data when a plain GROUP BY will not suffice. For instance, the preceding example could have included multiple AVG columns in its SELECT list, each with its own CASE expression. We might have had a query find the average salary for all employees in the salary ranges 0-2000 and 2000-5000. It would look like:

```
SELECT AVG(CASE WHEN e.sal BETWEEN 0 AND 2000 THEN e.sal ELSE null END) avg2000,
    AVG(CASE WHEN e.sal BETWEEN 2001 AND 5000 THEN e.sal ELSE null END) avg5000
 FROM emps e;
```

Although this query places the aggregates of independent subsets data into separate columns, by adding a CASE expression to the GROUP BY clause we can display the aggregates as the rows of a single column. The next section shows the flexibility of this approach with two approaches to creating histograms with CASE.

Creating Histograms

You can use the CASE statement when you want to obtain histograms with user-defined buckets (both in number of buckets and width of each bucket). The following are two examples of histograms created with CASE statements. In the first example, the histogram totals are shown in multiple columns and a single row is returned. In the second example, the histogram is shown with a label column and a single column for totals, and multiple rows are returned.

Example 22–28 Histogram Example 1

```
SELECT SUM(CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN 1 ELSE 0 END)
  AS "0-3999",
```

```
SUM(CASE WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN 1 ELSE 0 END)
  AS "4000-7999",
SUM(CASE WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN 1 ELSE 0 END)
  AS "8000-11999",
SUM(CASE WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN 1 ELSE 0 END)
  AS "12000-16000"
FROM customers WHERE cust_city = 'Marshal';

    0-3999  4000-7999 8000-11999 12000-16000
---------- ---------- ---------- -----------
         8          7          7           1
```

Example 22–29 Histogram Example 2

```
SELECT (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN  ' 0 - 3999'
   WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
   WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
   WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END)
  AS BUCKET, COUNT(*) AS Count_in_Group
FROM customers WHERE cust_city = 'Marshal' GROUP BY
 (CASE WHEN cust_credit_limit BETWEEN  0 AND 3999 THEN ' 0 - 3999'
 WHEN cust_credit_limit BETWEEN  4000 AND 7999 THEN ' 4000 - 7999'
 WHEN cust_credit_limit BETWEEN  8000 AND 11999 THEN  ' 8000 - 11999'
 WHEN cust_credit_limit BETWEEN  12000 AND 16000 THEN '12000 - 16000' END);

BUCKET        COUNT_IN_GROUP
------------- --------------
 0 - 3999                  8
 4000 - 7999               7
 8000 - 11999              7
12000 - 16000              1
```

Frequent Itemsets

Instead of counting how often a given event occurs (for example, how often someone has purchased milk at the grocery), you may find it useful to count how often multiple events occur together (for example, how often someone has purchased both milk and cereal together at the grocery store). You can count these multiple events using what is called a frequent itemset, which is, as the name implies, a set of items. Some examples of itemsets could be all of the products that a given customer purchased in a single trip to the grocery store (commonly called a market basket), the web pages that a user accessed in a single session, or the financial services that a given customer utilizes.

The practical motivation for using a frequent itemset is to find those itemsets that occur most often. If you analyze a grocery store's point-of-sale data, you might, for example, discover that milk and bananas are the most commonly bought pair of items. Frequent itemsets have thus been used in business intelligence environments for many years, with the most common one being for market basket analysis in the retail industry. Frequent itemset calculations are integrated with the database, operating on top of relational tables and accessed through SQL. This integration provides the following key benefits:

- Applications that previously relied on frequent itemset operations now benefit from significantly improved performance as well as simpler implementation.
- SQL-based applications that did not previously use frequent itemsets can now be easily extended to take advantage of this functionality.

Frequent itemsets analysis is performed with the PL/SQL package `DBMS_FREQUENT_ITEMSETS`. See *Oracle Database PL/SQL Packages and Types Reference* for more

information. In addition, there is an example of frequent itemset usage in "Frequent itemsets" on page 24-8.

23

SQL for Modeling

This chapter discusses using SQL modeling, and includes:

- Overview of SQL Modeling
- Basic Topics in SQL Modeling
- Advanced Topics in SQL Modeling
- Performance Considerations with SQL Modeling
- Examples of SQL Modeling

Overview of SQL Modeling

The `MODEL` clause brings a new level of power and flexibility to SQL calculations. With the `MODEL` clause, you can create a multidimensional array from query results and then apply formulas (called rules) to this array to calculate new values. The rules can range from basic arithmetic to simultaneous equations using recursion. For some applications, the `MODEL` clause can replace PC-based spreadsheets. Models in SQL leverage Oracle Database's strengths in scalability, manageability, collaboration, and security. The core query engine can work with unlimited quantities of data. By defining and executing models within the database, users avoid transferring large data sets to and from separate modeling environments. Models can be shared easily across workgroups, ensuring that calculations are consistent for all applications. Just as models can be shared, access can also be controlled precisely with Oracle's security features. With its rich functionality, the `MODEL` clause can enhance all types of applications.

The `MODEL` clause enables you to create a multidimensional array by mapping the columns of a query into three groups: partitioning, dimension, and measure columns. These elements perform the following tasks:

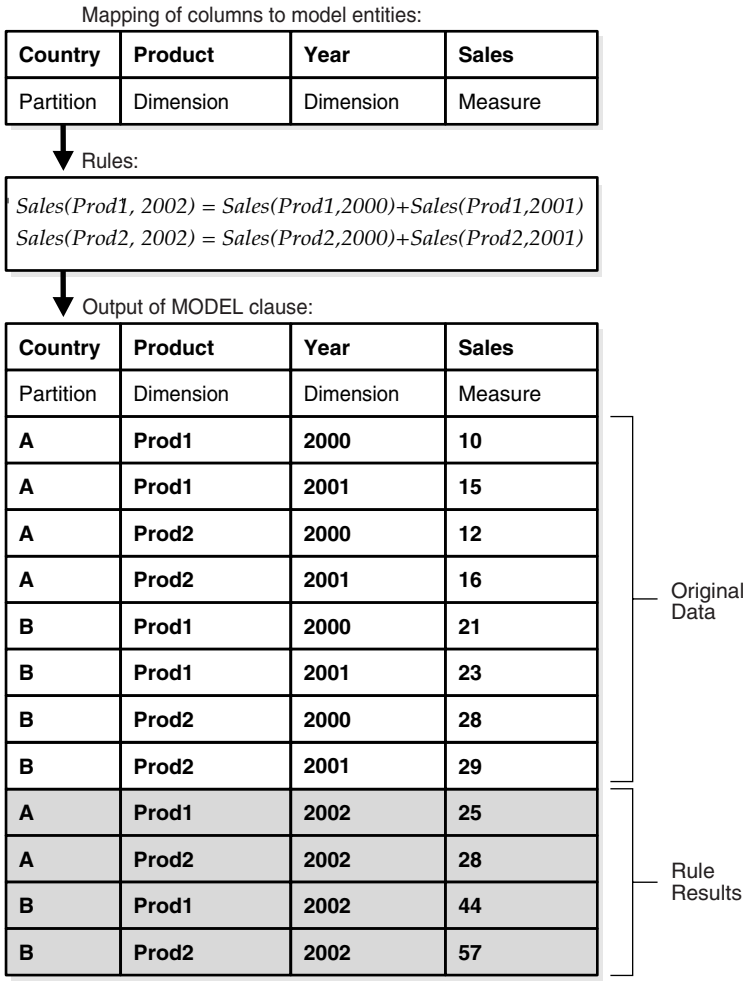
- Partition columns define the logical blocks of the result set in a way similar to the partitions of the analytical functions described in Chapter 22, "SQL for Analysis and Reporting". Rules in the `MODEL` clause are applied to each partition independent of other partitions. Thus, partitions serve as a boundary point for parallelizing the `MODEL` computation.
- Dimension columns define the multi-dimensional array and are used to identify cells within a partition. By default, a full combination of dimensions should identify just one cell in a partition. In default mode, they can be considered analogous to the key of a relational table.
- Measures are equivalent to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed

by specifying its full combination of dimensions. Note that each partition may have a cell that matches a given combination of dimensions.

The MODEL clause enables you to specify rules to manipulate the measure values of the cells in the multi-dimensional array defined by partition and dimension columns. Rules access and update measure column values by directly specifying dimension values. The references used in rules result in a highly readable model. Rules are concise and flexible, and can use wild cards and looping constructs for maximum expressiveness. Oracle Database evaluates the rules in an efficient way, parallelizes the model computation whenever possible, and provides a seamless integration of the MODEL clause with other SQL clauses. The MODEL clause, thus, is a scalable and manageable way of computing business models in the database.

Figure 23–1 offers a conceptual overview of the modeling feature of SQL. The figure has three parts. The top segment shows the concept of dividing a typical table into partition, dimension, and measure columns. The middle segment shows two rules that calculate the value of Prod1 and Prod2 for the year 2002. Finally, the third part shows the output of a query that applies the rules to such a table with hypothetical data. The unshaded output is the original data as it is retrieved from the database, while the shaded output shows the rows calculated by the rules. Note that results in partition A are calculated independently from results of partition B.

Figure 23–1 Model Elements



How Data is Processed in a SQL Model

Figure 23–2 shows the flow of processing within a simple `MODEL` clause. In this case, we will follow data through a `MODEL` clause that includes three rules. One of the rules updates an existing value, while the other two create new values for a forecast. The figure shows that the rows of data retrieved by a query are fed into the `MODEL` clause and rearranged into an array. Once the array is defined, rules are applied one by one to the data. The shaded cells in Figure 23–2 represent new data created by the rules and the cells enclosed by ovals represent the source data for the new values. Finally, the data, including both its updated values and newly created values, is rearranged into row form and presented as the results of the query. Note that no data is inserted into any table by this query.

Figure 23–2 Model Flow Processing

Why Use SQL Modeling?

Oracle modeling enables you to perform sophisticated calculations on your data. A typical case is when you want to apply business rules to data and then generate reports. Because Oracle Database integrates modeling calculations into the database, performance and manageability are enhanced significantly. Consider the following query:

```
SELECT SUBSTR(country, 1, 20) country,
      SUBSTR(product, 1, 15) product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
  MEASURES (sales sales)
  RULES
  (sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
   sales['Y Box', 2002] = sales['Y Box', 2001],
   sales['All_Products', 2002] = sales['Bounce', 2002] + sales['Y Box', 2002])
ORDER BY country, product, year;
```

This query partitions the data in `sales_view` (which is illustrated in "Base Schema" on page 23-7) on country so that the model computation, as defined by the three rules, is performed on each country. This model calculates the sales of Bounce in 2002 as the sum of its sales in 2000 and 2001, and sets the sales for Y Box in 2002 to the same value as they were in 2001. Also, it introduces a new product category All_Products (`sales_view` does not have the product All_Products) for year 2002 to be the sum of sales of Bounce and Y Box for that year. The output of this query is as follows, where bold text indicates new values:

```
COUNTRY              PRODUCT               YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                1999    2474.78
Italy                Bounce                2000    4333.69
Italy                Bounce                2001     4846.3
Italy                Bounce                2002    9179.99
...
Italy                Y Box                 1999   15215.16
Italy                Y Box                 2000   29322.89
Italy                Y Box                 2001   81207.55
Italy                Y Box                 2002   81207.55
...
Italy                All_Products          2002   90387.54
...
Japan                Bounce                1999     2961.3
Japan                Bounce                2000    5133.53
Japan                Bounce                2001     6303.6
Japan                Bounce                2002   11437.13
...
Japan                Y Box                 1999   22161.91
Japan                Y Box                 2000   45690.66
Japan                Y Box                 2001   89634.83
Japan                Y Box                 2002   89634.83
...
Japan                All_Products          2002  101071.96
...
```

Note that, while the sales values for Bounce and Y Box exist in the input, the values for All_Products are derived.

SQL Modeling Capabilities

Oracle Database provides the following capabilities with the `MODEL` clause:

- Cell addressing using dimension values

  Measure columns in individual rows are treated like cells in a multi-dimensional array and can be referenced and updated using dimension values. For example, in

a fact table `ft(country, year, sales)`, you can designate `country` and `year` to be dimension columns and `sales` to be the measure and reference sales for a given country and year as `sales[country='Spain', year=1999]`. This gives you the sales value for Spain in 1999. You can also use a shorthand form `sales['Spain', 1999]`, which has the same meaning. There are a few semantic differences between these notations, though. See "Cell Referencing" on page 23-11 for further details.

- Symbolic array computation

  You can specify a series of formulas, called rules, to operate on the data. Rules can invoke functions on individual cells or on a set or range of cells. An example involving individual cells is the following:

  ```
  sales[country='Spain',year=2001] = sales['Spain',2000]+ sales['Spain',1999]
  ```

  This sets the sales in Spain for the year 2001 to the sum of sales in Spain for 1999 and 2000. An example involving a range of cells is the following:

  ```
  sales[country='Spain',year=2001] =
     MAX(sales)['Spain',year BETWEEN 1997 AND 2000]
  ```

  This sets the sales in Spain for the year 2001 equal to the maximum sales in Spain between 1997 and 2000.

- `UPSERT`, `UPSERT ALL`, and `UPDATE` options

  Using the `UPSERT` option, which is the default, you can create cell values that do not exist in the input data. If the cell referenced exists in the data, it is updated. If the cell referenced does not exist in the data, and the rule uses appropriate notation, then the cell is inserted. The `UPSERT ALL` option enables you to have `UPSERT` behavior for a wider variety of rules. The `UPDATE` option, on the other hand, would never insert any new cells.

  You can specify these options globally, in which case they apply to all rules, or per each rule. If you specify an option at the rule level, it overrides the global option. Consider the following rules:

  ```
  UPDATE sales['Spain', 1999] = 3567.99,
  UPSERT sales['Spain', 2001] = sales['Spain', 2000]+ sales['Spain', 1999]
  ```

  The first rule updates the cell for sales in Spain for 1999. The second rule updates the cell for sales in Spain for 2001 if it exists, otherwise, it creates a new cell.

- Wildcard specification of dimensions

  You can use `ANY` and `IS ANY` to specify all values in a dimension. As an example, consider the following statement:

  ```
  sales[ANY, 2001] = sales['Japan', 2000]
  ```

  This rule sets the 2001 sales of all countries equal to the sales value of Japan for the year 2000. All values for the dimension, including nulls, satisfy the `ANY` specification. You can also specify this using an `IS ANY` predicate as in the following:

  ```
  sales[country IS ANY, 2001] = sales['Japan', 2000]
  ```

- Accessing dimension values using the `CV` function

  You can use the `CV` function on the right side of a rule to access the value of a dimension column of the cell referenced on the left side of a rule. It enables you to combine multiple rules performing similar computation into a single rule, thus

resulting in concise specification. For example, you can combine the following rules:

```
sales[country='Spain', year=2002] = 1.2 * sales['Spain', 2001],
sales[country='Italy', year=2002] = 1.2 * sales['Italy', 2001],
sales[country='Japan', year=2002] = 1.2 * sales['Japan', 2001]
```

They can be combined into one single rule:

```
sales[country IN ('Spain', 'Italy', 'Japan'), year=2002] = 1.2 *
   sales[CV(country), 2001]
```

Observe that the `CV` function passes the value for the `country` dimension from the left to the right side of the rule.

- Ordered computation

  For rules updating a set of cells, the result may depend on the ordering of dimension values. You can force a particular order for the dimension values by specifying an `ORDER BY` in the rule. An example is the following rule:

  ```
  sales[country IS ANY, year BETWEEN 2000 AND 2003] ORDER BY year =
    1.05 * sales[CV(country), CV(year)-1]
  ```

  This ensures that the years are referenced in increasing chronological order.

- Automatic rule ordering

  Rules in the `MODEL` clause can be automatically ordered based on dependencies among the cells using the `AUTOMATIC ORDER` keywords. For example, in the following assignments, the last two rules will be processed before the first rule because the first depends on the second and third:

  ```
  RULES AUTOMATIC ORDER
  {sales[c='Spain', y=2001] = sales[c='Spain', y=2000]
    + sales[c='Spain', y=1999]
  sales[c='Spain', y=2000] = 50000,
  sales[c='Spain', y=1999] = 40000}
  ```

- Iterative rule evaluation

  You can specify iterative rule evaluation, in which case the rules are evaluated iteratively until the termination condition is satisfied. Consider the following specification:

  ```
  MODEL DIMENSION BY (x) MEASURES (s)
    RULES ITERATE (4) (s[x=1] = s[x=1]/2)
  ```

  This statement specifies that the formula `s[x=1] = s[x=1]/2` evaluation be repeated four times. The number of iterations is specified in the `ITERATE` option of the `MODEL` clause. It is also possible to specify a termination condition by using an `UNTIL` clause.

  Iterative rule evaluation is an important tool for modeling recursive relationships between entities in a business application. For example, a loan amount might depend on the interest rate where the interest rate in turn depends on the amount of the loan.

- Reference models

  A model can include multiple ref models, which are read-only arrays. Rules can reference cells from different reference models. Rules can update or insert cells in only one multi-dimensional array, which is called the main model. The use of

reference models enables you to relate models with different dimensionality. For example, assume that, in addition to the fact table `ft(country, year, sales)`, you have a table with currency conversion ratios `cr(country, ratio)` with `country` as the dimension column and `ratio` as the measure. Each row in this table gives the conversion ratio of that country's currency to that of US dollar. These two tables could be used in rules such as the following:

```
dollar_sales['Spain',2001] = sales['Spain',2000] * ratio['Spain']
```

- Scalable computation

  You can partition data and evaluate rules within each partition independent of other partitions. This enables parallelization of model computation based on partitions. For example, consider the following model:

  ```
  MODEL PARTITION BY (country) DIMENSION BY (year) MEASURES (sales)
    (sales[year=2001] = AVG(sales)[year BETWEEN 1990 AND 2000]
  ```

  The data is partitioned by country and, within each partition, you can compute the sales in 2001 to be the average of sales in the years between 1990 and 2000. Partitions can be processed in parallel and this results in a scalable execution of the model.

Basic Topics in SQL Modeling

This section introduces some of the basic ideas and uses for models, and includes:

- Base Schema
- MODEL Clause Syntax
- Keywords in SQL Modeling
- Cell Referencing
- Rules
- Order of Evaluation of Rules
- Global and Local Keywords for Rules
- UPDATE, UPSERT, and UPSERT ALL Behavior
- Treatment of NULLs and Missing Cells
- Reference Models

Base Schema

This chapter's examples are based on the following view `sales_view`, which is derived from the `sh` sample schema.

```
CREATE VIEW sales_view AS
SELECT country_name country, prod_name product, calendar_year year,
  SUM(amount_sold) sales, COUNT(amount_sold) cnt,
  MAX(calendar_year) KEEP (DENSE_RANK FIRST ORDER BY SUM(amount_sold) DESC)
  OVER (PARTITION BY country_name, prod_name) best_year,
  MAX(calendar_year) KEEP (DENSE_RANK LAST ORDER BY SUM(amount_sold) DESC)
  OVER (PARTITION BY country_name, prod_name) worst_year
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND sales.prod_id = products.prod_id AND
  sales.cust_id =customers.cust_id AND customers.country_id=countries.country_id
GROUP BY country_name, prod_name, calendar_year;
```

This query computes SUM and COUNT aggregates on the sales data grouped by country, product, and year. It will report for each product sold in a country, the year when the sales were the highest for that product in that country. This is called the best_year of the product. Similarly, worst_year gives the year when the sales were the lowest.

MODEL Clause Syntax

The MODEL clause enables you to define multi-dimensional calculations on the data in the SQL query block. In multi-dimensional applications, a fact table consists of columns that uniquely identify a row with the rest serving as dependent measures or attributes. The MODEL clause lets you specify the PARTITION, DIMENSION, and MEASURE columns that define the multi-dimensional array, the rules that operate on this multi-dimensional array, and the processing options.

The MODEL clause contains a list of updates representing array computation within a partition and is a part of a SQL query block. Its structure is as follows:

```
MODEL
[<global reference options>]
[<reference models>]
[MAIN <main-name>]
  [PARTITION BY (<cols>)]
  DIMENSION BY (<cols>)
  MEASURES (<cols>)
  [<reference options>]
  [RULES]  <rule options>
  (<rule>, <rule>,.., <rule>)
  <global reference options> ::= <reference options> <ret-opt>
   <ret-opt> ::= RETURN {ALL|UPDATED} ROWS
  <reference options> ::=
  [IGNORE NAV | [KEEP NAV]
  [UNIQUE DIMENSION | UNIQUE SINGLE REFERENCE]
  <rule options> ::=
  [UPDATE | UPSERT | UPSERT ALL]
  [AUTOMATIC ORDER | SEQUENTIAL ORDER]
  [ITERATE (<number>)  [UNTIL <condition>]]
  <reference models> ::= REFERENCE ON <ref-name> ON (<query>)
  DIMENSION BY (<cols>) MEASURES (<cols>) <reference options>
```

Each rule represents an assignment. Its left side references a cell or a set of cells and the right side can contain expressions involving constants, host variables, individual cells or aggregates over ranges of cells. For example, consider Example 23–1.

Example 23–1 Simple Query with the MODEL Clause

```
SELECT SUBSTR(country,1,20) country, SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE country in ('Italy', 'Japan')
MODEL
  RETURN UPDATED ROWS
  MAIN simple_model
  PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales)
  RULES
   (sales['Bounce', 2001] = 1000,
    sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
    sales['Y Box', 2002] = sales['Y Box', 2001])
```

```
ORDER BY country, product, year;
```

This query defines model computation on the rows from sales_view for the countries Italy and Japan. This model has been given the name simple_model. It partitions the data on country and defines, within each partition, a two-dimensional array on product and year. Each cell in this array holds the value of the sales measure. The first rule of this model sets the sales of Bounce in year 2001 to 1000. The next two rules define that the sales of Bounce in 2002 are the sum of its sales in years 2001 and 2000, and the sales of Y Box in 2002 are same as that of the previous year 2001.

Specifying RETURN UPDATED ROWS makes the preceding query return only those rows that are updated or inserted by the model computation. By default or if you use RETURN ALL ROWS, you would get all rows not just the ones updated or inserted by the MODEL clause. The query produces the following output:

```
COUNTRY              PRODUCT               YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2001       1000
Italy                Bounce                2002    5333.69
Italy                Y Box                 2002   81207.55
Japan                Bounce                2001       1000
Japan                Bounce                2002    6133.53
Japan                Y Box                 2002   89634.83
```

Note that the MODEL clause does not update or insert rows into database tables. The following query illustrates this by showing that sales_view has not been altered:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(product,1,15) product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan');

COUNTRY              PRODUCT               YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                1999    2474.78
Italy                Bounce                2000    4333.69
Italy                Bounce                2001     4846.3
...
```

Observe that the update of the sales value for Bounce in the 2001 done by this MODEL clause is not reflected in the database. If you want to update or insert rows in the database tables, you should use the INSERT, UPDATE, or MERGE statements.

In the preceding example, columns are specified in the PARTITION BY, DIMENSION BY, and MEASURES list. You can also specify constants, host variables, single-row functions, aggregate functions, analytical functions, or expressions involving them as partition and dimension keys and measures. However, you must alias them in PARTITION BY, DIMENSION BY, and MEASURES lists. You must use aliases to refer these expressions in the rules, SELECT list, and the query ORDER BY. The following example shows how to use expressions and aliases:

```
SELECT country, p product, year, sales, profits
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL
  RETURN UPDATED ROWS
  PARTITION BY (SUBSTR(country,1,20) AS country)
  DIMENSION BY (product AS p, year)
  MEASURES (sales, 0 AS profits)
  RULES
   (profits['Bounce', 2001] = sales['Bounce', 2001] * 0.25,
```

```
    sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
    profits['Bounce', 2002] = sales['Bounce', 2002] * 0.35)
ORDER BY country, year;

COUNTRY   PRODUCT    YEAR   SALES         PROFITS
-------   ---------  ----   --------     --------
Italy     Bounce     2001     4846.3     1211.575
Italy     Bounce     2002    9179.99    3212.9965
Japan     Bounce     2001     6303.6       1575.9
Japan     Bounce     2002   11437.13    4002.9955
```

Note that the alias "0 AS profits" initializes all cells of the profits measure to 0. See *Oracle Database SQL Language Reference* for more information regarding `MODEL` clause syntax.

Keywords in SQL Modeling

This section defines keywords used in SQL modeling.

Assigning Values and Null Handling

- `UPSERT`

  This updates the measure values of existing cells. If the cells do not exist, and the rule has appropriate notation, they are inserted. If any of the cell references are symbolic, no cells are inserted.

- `UPSERT ALL`

  This is similar to `UPSERT`, except it allows a broader set of rule notation to insert new cells.

- `UPDATE`

  This updates existing cell values. If the cell values do not exist, no updates are done.

- `IGNORE NAV`

  For numeric cells, this treats values that are not available as 0. This means that a cell not supplied to `MODEL` by the query result set will be treated as a zero for the calculation. This can be used at a global level for all measures in a model.

- `KEEP NAV`

  This keeps cell values that are not available unchanged. It is useful for making exceptions when `IGNORE NAV` is specified at the global level. This is the default, and can be omitted.

Calculation Definition

- `MEASURES`

  The set of values that are modified or created by the model.

- `RULES`

  The expressions that assign values to measures.

- `AUTOMATIC ORDER`

  This causes all rules to be evaluated in an order based on their logical dependencies.

- `SEQUENTIAL ORDER`

This causes rules to be evaluated in the order they are written. This is the default.

- UNIQUE DIMENSION

  This is the default, and it means that the combination of PARTITION BY and DIMENSION BY columns in the MODEL clause must uniquely identify each and every cell in the model. This uniqueness is explicitly verified at query execution when necessary, in which case it may increase processing time.

- UNIQUE SINGLE REFERENCE

  The PARTITION BY and DIMENSION BY clauses uniquely identify single point references on the right-hand side of the rules. This may reduce processing time by avoiding explicit checks for uniqueness at query execution.

- RETURN [ALL|UPDATED] ROWS

  This enables you to specify whether to return all rows selected or only those rows updated by the rules. The default is ALL, while the alternative is UPDATED ROWS.

Cell Referencing

In the MODEL clause, a relation is treated as a multi-dimensional array of cells. A cell of this multi-dimensional array contains the measure values and is indexed using DIMENSION BY keys, within each partition defined by the PARTITION BY keys. For example, consider the following:

```
SELECT country, product, year, sales, best_year, best_year
FROM sales_view
MODEL
  PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales, best_year)
  (<rules> ..)
ORDER BY country, product, year;
```

This partitions the data by country and defines within each partition, a two-dimensional array on product and year. The cells of this array contain two measures: sales and best_year.

Accessing the measure value of a cell by specifying the DIMENSION BY keys constitutes a cell reference. An example of a cell reference is as follows:

```
sales[product= 'Bounce', year=2000]
```

Here, we are accessing the sales value of a cell referenced by product Bounce and the year 2000. In a cell reference, you can specify DIMENSION BY keys either symbolically as in the preceding cell reference or positionally as in sales['Bounce', 2000].

Symbolic Dimension References

A symbolic dimension reference (or symbolic reference) is one in which DIMENSION BY key values are specified with a boolean expression. For example, the cell reference sales[year >= 2001] has a symbolic reference on the DIMENSION BY key year and specifies all cells whose year value is greater than or equal to 2001. An example of symbolic references on product and year dimensions is sales[product = 'Bounce', year >= 2001].

Positional Dimension References

A positional dimension reference (or positional reference, in short) is a constant or a constant expression specified for a dimension. For example, the cell reference

sales['Bounce'] has a positional reference on the product dimension and accesses sales value for the product Bounce. The constants (or constant expressions) in a cell reference are matched to the column order specified for DIMENSION BY keys. The following example shows the usage of positional references on dimensions:

```
sales['Bounce', 2001]
```

Assuming DIMENSION BY keys to be product and year in that order, it accesses the sales value for Bounce and 2001.

Based on how they are specified, cell references are either single cell or multi-cell reference.

Rules

Model computation is expressed in rules that manipulate the cells of the multi-dimensional array defined by PARTITION BY, DIMENSION BY, and MEASURES clauses. A rule is an assignment statement whose left side represents a cell or a range of cells and whose right side is an expression involving constants, bind variables, individual cells or an aggregate function on a range of cells. Rules can use wild cards and looping constructs for maximum expressiveness. An example of a rule is the following:

```
sales['Bounce', 2003] = 1.2 * sales['Bounce', 2002]
```

This rule says that, for the product Bounce, the sales for 2003 are 20% more than that of 2002.

Note that this rule refers to single cells on both the left and right side and is relatively simple. Complex rules can be written with multi-cell references, aggregates, and nested cell references. These are discussed in the following sections.

Single Cell References

This type of rule involves single cell reference on the left side with constants and single cell references on the right side. Some examples are the following:

```
sales[product='Finding Fido', year=2003] = 100000
sales['Bounce', 2003] = 1.2 * sales['Bounce', 2002]
sales[product='Finding Fido', year=2004] = 0.8 * sales['Standard Mouse Pad',
  year=2003] + sales['Finding Fido', 2003]
```

Multi-Cell References on the Right Side

Multi-cell references can be used on the right side of rules, in which case an aggregate function needs to be applied on them to convert them to a single value. All existing aggregate functions including analytic aggregates (inverse percentile functions, hypothetical rank and distribution functions and so on) and statistical aggregates (correlation, regression slope and so on), and user-defined aggregate functions can be used. Windowing functions such as RANK and MOVING_AVG can be used as well. For example, the rule to compute the sales of Bounce for 2003 to be 100 more than the maximum sales in the period 1998 to 2002 would be:

```
sales['Bounce', 2003] = 100 + MAX(sales)['Bounce', year BETWEEN 1998 AND 2002]
```

The following example illustrates the usage of inverse percentile function PERCENTILE_DISC. It projects Finding Fido sales for year 2003 to be 30% more than the median sales for products Finding Fido, Standard Mouse Pad, and Boat for all years prior to 2003.

```
sales[product='Finding Fido', year=2003] = 1.3 *
```

```
  PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY sales) [product IN ('Finding
  Fido','Standard Mouse Pad','Boat'), year < 2003]
```

Aggregate functions can appear only on the right side of rules. Arguments to the aggregate function can be constants, bind variables, measures of the `MODEL` clause, or expressions involving them. For example, the rule computes the sales of Bounce for 2003 to be the weighted average of its sales for years from 1998 to 2002 would be:

```
sales['Bounce', 2003] =
   AVG(sales * weight)['Bounce', year BETWEEN 1998 AND 2002]
```

Multi-Cell References on the Left Side

Rules can have multi-cell references on the left side as in the following:

```
sales['Standard Mouse Pad', year > 2000] =
   0.2 * sales['Finding Fido', year=2000]
```

This rule accesses a range of cells on the left side (cells for product Standard Mouse Pad and year greater than 2000) and assigns sales measure of each such cell to the value computed by the right side expression. Computation by the preceding rule is described as "sales of Standard Mouse Pad for years after 2000 is 20% of the sales of Finding Fido for year 2000". This computation is simple in that the right side cell references and hence the right side expression are the same for all cells referenced on the left.

Use of the CV Function

The use of the `CV` function provides the capability of relative indexing where dimension values of the cell referenced on the left side are used on the right side cell references. The `CV` function takes a dimension key as its argument, so it provides the value of a `DIMENSION BY` key of the cell currently referenced on the left side. As an example, consider the following:

```
sales[product='Standard Mouse Pad', year>2000] =
  sales[CV(product), CV(year)] + 0.2 * sales['Finding Fido', 2000]
```

When the left side references the cell Standard Mouse Pad and 2001, the right side expression would be:

```
sales['Standard Mouse Pad', 2001] + 0.2 * sales['Finding Fido', 2000]
```

Similarly, when the left side references the cell Standard Mouse Pad and 2002, the right side expression we would evaluate is:

```
sales['Standard Mouse Pad', 2002] + 0.2 * sales['Finding Fido', 2000]
```

It is also possible to use `CV` without any argument as in `CV()` and in which case, positional referencing is implied. `CV()` may be used outside a cell reference, but when used in this way its argument must contain the name of the dimension desired. You can also write the preceding rule as:

```
sales[product='Standard Mouse Pad', year>2000] =
  sales[CV(), CV()] + 0.2 * sales['Finding Fido', 2000]
```

The first `CV()` reference corresponds to `CV(product)` and the latter corresponds to `CV(year)`. The `CV` function can be used only in right side cell references. Another example of the usage of `CV` function is the following:

```
sales[product IN ('Finding Fido','Standard Mouse Pad','Bounce'), year
  BETWEEN 2002 AND 2004] = 2 * sales[CV(product), CV(year)-10]
```

This rule says that, for products Finding Fido, Standard Mouse Pad, and Bounce, the sales for years between 2002 and 2004 will be twice of what their sales were 10 years ago.

Use of the ANY Wildcard

You can use the wild card ANY in cell references to match all dimension values including nulls. ANY may be used on both the left and right side of rules. For example, a rule for the computation "sales of all products for 2003 are 10% more than their sales for 2002" would be the following:

```
sales[product IS ANY, 2003] = 1.1 * sales[CV(product), 2002]
```

Using positional references, it can also be written as:

```
sales[ANY, 2003] = 1.1 * sales[CV(), 2002]
```

Note that ANY is treated as a symbolic reference even if it is specified positionally, because it really means that (dimension IS NOT NULL OR dimension IS NULL).

Nested Cell References

Cell references can be nested. In other words, cell references providing dimension values can be used within a cell reference. An example, assuming best_year is a measure, for nested cell reference is given as follows:

```
sales[product='Bounce', year = best_year['Bounce', 2003]]
```

Here, the nested cell reference best_year['Bounce', 2003] provides value for the dimension key year and is used in the symbolic reference for year. Measures best_year and worst_year give, for each year (y) and product (p) combination, the year for which sales of product p were highest or lowest. The following rule computes the sales of Standard Mouse Pad for 2003 to be the average of Standard Mouse Pad sales for the years in which Finding Fido sales were highest and lowest:

```
sales['Standard Mouse Pad', 2003] = (sales[CV(), best_year['Finding Fido',
  CV(year)]] + sales[CV(), worst_year['Finding Fido', CV(year)]]) / 2
```

Oracle Database allows only one level of nesting, and only single cell references can be used as nested cell references. Aggregates on multi-cell references cannot be used in nested cell references.

Order of Evaluation of Rules

By default, rules are evaluated in the order they appear in the MODEL clause. You can specify an optional keyword SEQUENTIAL ORDER in the MODEL clause to make such an evaluation order explicit. SQL models with sequential rule order of evaluation are called sequential order models. For example, the following RULES specification makes Oracle Database evaluate rules in the specified sequence:

```
RULES SEQUENTIAL ORDER
 (sales['Bounce', 2001] =
    sales['Bounce', 2000] + sales['Bounce', 1999],   --Rule R1
  sales['Bounce', 2000] = 50000,                     --Rule R2
  sales['Bounce', 1999] = 40000)                     --Rule R3
```

Alternatively, the option AUTOMATIC ORDER enables Oracle Database to determine the order of evaluation of rules automatically. Oracle examines the cell references within

rules and finds dependencies among rules. If cells referenced on the left side of rule R1 are referenced on the right side of another rule R2, then R2 is considered to depend on R1. In other words, rule R1 should be evaluated before rule R2. If you specify AUTOMATIC ORDER in the preceding example as in:

```
RULES AUTOMATIC ORDER
 (sales['Bounce', 2001] = sales['Bounce', 2000] + sales['Bounce', 1999],
  sales['Bounce', 2000] = 50000,
  sales['Bounce', 1999] = 40000)
```

Rules 2 and 3 are evaluated, in some arbitrary order, before rule 1. This is because rule 1 depends on rules 2 and 3 and hence need to be evaluated after rules 2 and 3. The order of evaluation among second and third rules can be arbitrary as they do not depend on one another. The order of evaluation among rules independent of one another can be arbitrary. SQL models with an automatic order of evaluation, as in the preceding fragment, are called automatic order models.

In an automatic order model, multiple assignments to the same cell are not allowed. In other words, measure of a cell can be assigned only once. Oracle Database will return an error in such cases as results would be non-deterministic. For example, the following rule specification will generate an error as sales['Bounce', 2001] is assigned more than once:

```
RULES AUTOMATIC ORDER
 (sales['Bounce', 2001] = sales['Bounce', 2000] + sales['Bounce', 1999],
  sales['Bounce', 2001] = 50000,
  sales['Bounce', 2001] = 40000)
```

The rules assigning the sales of product Bounce for 2001 do not depend on one another and hence, no particular evaluation order can be fixed among them. This leads to non-deterministic results as the evaluation order is arbitrary - sales['Bounce', 2001] can be 40000 or 50000 or sum of Bounce sales for years 1999 and 2000. Oracle Database prevents this by disallowing multiple assignments when AUTOMATIC ORDER is specified. However, multiple assignments are fine in sequential order models. If SEQUENTIAL ORDER was specified instead of AUTOMATIC ORDER in the preceding example, the result of sales['Bounce', 2001] would be 40000.

Global and Local Keywords for Rules

You can specify an UPDATE, UPSERT, UPSERT ALL, IGNORE NAV, and KEEP NAV option at the global level in the RULES clause in which case all rules operate in the respective mode. These options can be specified at a local level with each rule and in which case, they override the global behavior. For example, in the following specification:

```
RULES UPDATE
(UPDATE s['Bounce',2001] = sales['Bounce',2000] + sales['Bounce',1999],
 UPSERT s['Y Box', 2001] = sales['Y Box', 2000] + sales['Y Box', 1999],
   sales['Mouse Pad', 2001] = sales['Mouse Pad', 2000] +
   sales['Mouse Pad',1999])
```

The UPDATE option is specified at the global level so, the first and third rules operate in update mode. The second rule operates in upsert mode as an UPSERT keyword is specified with that rule. Note that no option was specified for the third rule and hence it inherits the update behavior from the global option.

UPDATE, UPSERT, and UPSERT ALL Behavior

You can determine how cells in rules behave by choosing whether to have UPDATE, UPSERT, or UPSERT ALL semantics. By default, rules in the MODEL clause have UPSERT semantics, though you can specify an optional UPSERT keyword to make the upsert semantic explicit.

The following sections discuss these three types of behavior:

- UPDATE Behavior
- UPSERT Behavior
- UPSERT ALL Behavior

UPDATE Behavior

The UPDATE option forces strict update mode. In this mode, the rule is ignored if the cell it references on the left side does not exist. If the cell referenced on the left side of a rule exists, then its measure is updated with the value of the right side expression. Otherwise, if a cell reference is positional, a new cell is created (that is, inserted into the multi-dimensional array) with the measure value equal to the value of the right side expression. If a cell reference is not positional, it will not insert cells. Note that if there are any symbolic references in a cell's specification, inserts are not possible in an upsert rule. For example, consider the following rule:

```
sales['Bounce', 2003] = sales['Bounce', 2001] + sales ['Bounce', 2002]
```

The cell for product Bounce and year 2003, if it exists, gets updated with the sum of Bounce sales for years 2001 and 2002, otherwise, it gets created. If you had created the same rule using any symbolic references, no updates would be performed, as in the following:

```
sales[prod= 'Bounce', year= 2003] = sales['Bounce', 2001] + sales ['Bounce', 2002]
```

UPSERT Behavior

Using UPSERT creates a new cell corresponding to the one referenced on the left side of the rule when the cell is missing, and the cell reference contains only positional references qualified by constants. Note that cell references created with FOR loops (described in "Advanced Topics in SQL Modeling" on page 23-23) are treated as positional references, so the values FOR loops create will be used to insert new cells. Assuming you do not have cells for years greater than 2003, consider the following rule:

```
UPSERT sales['Bounce', year = 2004] = 1.1 * sales['Bounce', 2002]
```

This would not create any new cell because of the symbolic reference year = 2004. However, consider the following:

```
UPSERT sales['Bounce', 2004] = 1.1 * sales['Bounce', 2002]
```

This would create a new cell for product Bounce for year 2004. On a related note, new cells will not be created if any of the references is ANY. This is because ANY is a predicate that qualifies all dimensional values including NULL. If there is a reference ANY for a dimension d, then it means the same thing as the predicate (d IS NOT NULL OR d IS NULL).

If an UPSERT rule uses FOR loops in its left side cell references, the list of upsert cells is generated by performing a cross product of all the distinct values for each dimension. Although UPSERT with FOR loops can be used to densify dimensions (see "Data

Densification for Reporting" on page 22-37), it is generally preferable to densify using the partition outer join operation.

UPSERT ALL Behavior

UPSERT ALL behavior allows model rules with existential predicates (comparisons, IN, ANY, and so on) in their left side to have UPSERT behavior. As an example, the following uses ANY and creates Bay Area as the combination of San Francisco, San Jose, and Oakland:

```
SELECT product, time, city, s sales
FROM cube_subquery
MODEL PARTITION BY (product)
DIMENSION BY (time, city) MEASURES(sales s
RULES UPSERT ALL
(s[ANY, 'Bay Area'] =
   s[CV(), 'San Francisco'] + s[CV(), 'San Jose'] + s[CV(), 'Oakland']
s['2004', ANY] = s['2002', CV()] + s['2003', CV()]);
```

In this example, the first rule simply inserts a Bay Area cell for each distinct time value, and the second rule inserts a 2004 cell for each distinct city value including Bay Area. This example is relatively simple as the existential predicates used on the left side are ANY predicates, but you can also use UPSERT ALL with more complex calculations.

It is important to understand exactly what the UPSERT ALL operation does, especially in cases where there is more than one symbolic dimension reference. Note that the behavior is different than the behavior of an UPSERT rule that uses FOR loops. When evaluating an UPSERT ALL rule, Oracle Database performs the following steps to create a list of cell references to be upserted.

Step 1 Find Cells

Find the existing cells that satisfy all the symbolic predicates of the cell reference.

Step 2 Find Distinct Dimension Values

Using just the dimensions that have symbolic references, find the distinct dimension value combinations of these cells.

Step 3 Perform a Cross Product

Perform a cross product of these value combinations with the dimension values specified through positional references.

Step 4 Upsert New Cells

The results of Step 3 are then used to upsert new cells into the array.

To illustrate these four steps, here is a brief example using abstracted data and a model with three dimensions. Consider a model dimensioned by (product, time, city) with a measure called sales. We wish to upsert new sales values for the city of z, and these sales values are copied from those of the city of y.

```
UPSERT ALL sales[ANY, ANY, 'z']= sales[CV(product),CV(time),'y']
```

Our source data set has these four rows:

```
PROD    TIME   CITY   SALES
   1    2002      x      10
   1    2003      x      15
   2    2002      y      21
```

```
   2    2003      y      24
```

The following explains the details of the four steps, applied to this data:

1. Because the symbolic predicates of the rule are ANY, any of the rows shown in this example is acceptable.
2. The distinct dimension combinations of cells with symbolic predicates that match the condition are: (1, 2002), (1, 2003), (2, 2002), and (2, 2003).
3. We find the cross product of these dimension combinations with the cells specified with positional references. In this case, it is simply a cross product with the value z, and the resulting cell references are: (1, 2002, z), (1, 2003, z), (2, 2002, z), and (2, 2003, z).
4. The cells listed in Step 3 will be upserted, with sales calculated based on the city y. Because there are no values for product 1 in city y, those cells created for product 1 will have NULL as their sales value. Of course, a different rule might have generated non-NULL results for all the new cells. Our result set includes the four original rows plus four new rows:

```
PROD    TIME   CITY    SALES
   1    2002      x       10
   1    2003      x       15
   2    2002      y       21
   2    2003      y       24
   1    2002      z     NULL
   1    2003      z     NULL
   2    2002      z       21
   2    2003      z       24
```

It is important to note that these results are not a cross product using all values of all dimensions. If that were the case, we would have cells such as (1,2002, y) and (2,2003, x). Instead, the results here are created using dimension combinations found in existing rows.

Treatment of NULLs and Missing Cells

Applications using models would not only have to deal with non-deterministic values for a cell measure in the form of NULL, but also with non-determinism in the form of missing cells. A cell, referenced by a single cell reference, that is missing in the data is called a missing cell. The MODEL clause provides a default treatment for nulls and missing cells that is consistent with the ANSI SQL standard and also provides options to treat them in other useful ways according to business logic, for example, to treat nulls as zero for arithmetic operations.

By default, NULL cell measure values are treated the same way as nulls are treated elsewhere in SQL. For example, in the following rule:

```
sales['Bounce', 2001] = sales['Bounce', 1999] + sales['Bounce', 2000]
```

The right side expression would evaluate to NULL if Bounce sales for one of the years 1999 and 2000 is NULL. Similarly, aggregate functions in rules would treat NULL values in the same way as their regular behavior where NULL values are ignored during aggregation.

Missing cells are treated as cells with NULL measure values. For example, in the preceding rule, if the cell for Bounce and 2000 is missing, then it is treated as a NULL value and the right side expression would evaluate to NULL.

Distinguishing Missing Cells from NULLs

The functions PRESENTV and PRESENTNNV enable you to identify missing cells and distinguish them from NULL values. These functions take a single cell reference and two expressions as arguments as in PRESENTV(cell, expr1, expr2). PRESENTV returns the first expression expr1 if the cell cell is existent in the data input to the MODEL clause. Otherwise, it returns the second expression expr2. For example, consider the following:

```
PRESENTV(sales['Bounce', 2000], 1.1*sales['Bounce', 2000], 100)
```

If the cell for product Bounce and year 2000 exists, it returns the corresponding sales multiplied by 1.1, otherwise, it returns 100. Note that if sales for the product Bounce for year 2000 is NULL, the preceding specification would return NULL.

The PRESENTNNV function not only checks for the presence of a cell but also whether it is NULL or not. It returns the first expression expr1 if the cell exists and is not NULL, otherwise, it returns the second expression expr2. For example, consider the following:

```
PRESENTNNV(sales['Bounce', 2000], 1.1*sales['Bounce', 2000], 100)
```

This would return 1.1*sales['Bounce', 2000] if sales['Bounce', 2000] exists and is not NULL. Otherwise, it returns 100.

Applications can use the IS PRESENT predicate in their model to check the presence of a cell in an explicit fashion.This predicate returns TRUE if cell exists and FALSE otherwise. The preceding example using PRESENTNNV can be written using IS PRESENT as:

```
CASE WHEN sales['Bounce', 2000] IS PRESENT AND sales['Bounce', 2000] IS NOT NULL
THEN 1.1 * sales['Bounce', 2000]
ELSE 100
END
```

The IS PRESENT predicate, like the PRESENTV and PRESENTNNV functions, checks for cell existence in the input data, that is, the data as existed before the execution of the MODEL clause. This enables you to initialize multiple measures of a cell newly inserted by an UPSERT rule. For example, if you want to initialize sales and profit values of a cell, if it does not exist in the data, for product Bounce and year 2003 to 1000 and 500 respectively, you can do so by the following:

```
RULES
 (UPSERT sales['Bounce', 2003] =
    PRESENTV(sales['Bounce', 2003], sales['Bounce', 2003], 1000),
  UPSERT profit['Bounce', 2003] =
    PRESENTV(profit['Bounce', 2003], profit['Bounce', 2003], 500))
```

The PRESENTV functions used in this formulation return TRUE or FALSE based on the existence of the cell in the input data. If the cell for Bounce and 2003 gets inserted by one of the rules, based on their evaluation order, PRESENTV function in the other rule would still evaluate to FALSE. You can consider this behavior as a preprocessing step to rule evaluation that evaluates and replaces all PRESENTV and PRESENTNNV functions and IS PRESENT predicate by their respective values.

Use Defaults for Missing Cells and NULLs

The MODEL clause, by default, treats missing cells as cells with NULL measure values. An optional KEEP NAV keyword can be specified in the MODEL clause to get this behavior.

If your application wants to default missing cells and nulls to some values, you can do so by using IS PRESENT, IS NULL predicates and PRESENTV, PRESENTNNV functions. But it may become cumbersome if you have lot of single cell references and rules. You can use IGNORE NAV option instead of the default KEEP NAV option to default nulls and missing cells to:

- 0 for numeric data
- Empty string for character/string data
- 01-JAN-2001 for data type data
- NULL for all other data types

Consider the following query:

```
SELECT product, year, sales
FROM sales_view
WHERE country = 'Poland'
MODEL
  DIMENSION BY (product, year) MEASURES (sales sales) IGNORE NAV
  RULES UPSERT
  (sales['Bounce', 2003] = sales['Bounce', 2002] + sales['Bounce', 2001]);
```

In this, the input to the MODEL clause does not have a cell for product Bounce and year 2002. Because of IGNORE NAV option, sales['Bounce', 2002] value would default to 0 (as sales is of numeric type) instead of NULL. Thus, sales['Bounce', 2003] value would be same as that of sales['Bounce', 2001].

Using NULLs in a Cell Reference

To use NULL values in a cell reference, you must use one of the following:

- Positional reference using wild card ANY as in sales[ANY].
- Symbolic reference using the IS ANY predicate as in sales[product IS ANY].
- Positional reference of NULL as in sales[NULL].
- Symbolic reference using IS NULL predicate as in sales[product IS NULL].

Note that symbolic reference sales[product = NULL] would not test for nulls in the product dimension. This behavior conforms with the standard handling of nulls by SQL.

Reference Models

In addition to the multi-dimensional array on which rules operate, which is called the main model, one or more read-only multi-dimensional arrays, called reference models, can be created and referenced in the MODEL clause to act as look-up tables for the main model. Like the main model, a reference model is defined over a query block and has DIMENSION BY and MEASURES clauses to indicate its dimensions and measures respectively. A reference model is created by the following subclause:

```
REFERENCE model_name ON (query) DIMENSION BY (cols) MEASURES (cols)
   [reference options]
```

Like the main model, a multi-dimensional array for the reference model is built before evaluating the rules. But, unlike the main model, reference models are read-only in that their cells cannot be updated and no new cells can be inserted after they are built. Thus, the rules in the main model can access cells of a reference model, but they cannot update or insert new cells into the reference model. The following is an example using a currency conversion table as a reference model:

```
CREATE TABLE dollar_conv_tbl(country VARCHAR2(30), exchange_rate NUMBER);
INSERT INTO dollar_conv_tbl VALUES('Poland', 0.25);
INSERT INTO dollar_conv_tbl VALUES('France', 0.14);
...
```

Now, to convert the projected sales of Poland and France for 2003 to the US dollar, you can use the dollar conversion table as a reference model as in the following:

```
SELECT country, year, sales, dollar_sales
FROM sales_view
GROUP BY country, year
MODEL
  REFERENCE conv_ref ON (SELECT country, exchange_rate FROM dollar_conv_tbl)
    DIMENSION BY (country) MEASURES (exchange_rate) IGNORE NAV
  MAIN conversion
    DIMENSION BY (country, year)
    MEASURES (SUM(sales) sales, SUM(sales) dollar_sales) IGNORE NAV
RULES
(dollar_sales['France', 2003] = sales[CV(country), 2002] * 1.02 *
   conv_ref.exchange_rate['France'],
   dollar_sales['Poland', 2003] =
      sales['Poland', 2002] * 1.05 * exchange_rate['Poland']);
```

Observe in this example that:

- A one dimensional reference model named `conv_ref` is created on rows from the table `dollar_conv_tbl` and that its measure `exchange_rate` has been referenced in the rules of the main model.
- The main model (called `conversion`) has two dimensions, country and year, whereas the reference model `conv_ref` has one dimension, country.
- Different styles of accessing the `exchange_rate` measure of the reference model. For France, it is rather explicit with `model_name.measure_name` notation `conv_ref.exchange_rate`, whereas for Poland, it is a simple `measure_name` reference `exchange_rate`. The former notation needs to be used to resolve any ambiguities in column names across main and reference models.

Growth rates, in this example, are hard coded in the rules. The growth rate for France is 2% and that of Poland is 5%. But they could come from a separate table and you can have a reference model defined on top of that. Assume that you have a `growth_rate(country, year, rate)` table defined as the following:

```
CREATE TABLE growth_rate_tbl(country VARCHAR2(30),
   year NUMBER, growth_rate NUMBER);
INSERT INTO growth_rate_tbl VALUES('Poland', 2002, 2.5);
INSERT INTO growth_rate_tbl VALUES('Poland', 2003, 5);
...
INSERT INTO growth_rate_tbl VALUES('France', 2002, 3);
INSERT INTO growth_rate_tbl VALUES('France', 2003, 2.5);
```

Then the following query computes the projected sales in dollars for 2003 for all countries:

```
SELECT country, year, sales, dollar_sales
FROM sales_view
GROUP BY country, year
MODEL
  REFERENCE conv_ref ON
          (SELECT country, exchange_rate FROM dollar_conv_tbl)
           DIMENSION BY (country c) MEASURES (exchange_rate) IGNORE NAV
  REFERENCE growth_ref ON
```

```
        (SELECT country, year, growth_rate FROM growth_rate_tbl)
         DIMENSION BY (country c, year y) MEASURES (growth_rate) IGNORE NAV
  MAIN projection
   DIMENSION BY (country, year) MEASURES (SUM(sales) sales, 0 dollar_sales)
  IGNORE NAV
  RULES
  (dollar_sales[ANY, 2003] = sales[CV(country), 2002] *
   growth_rate[CV(country), CV(year)] *
   exchange_rate[CV(country)]);
```

This query shows the capability of the `MODEL` clause in dealing with and relating objects of different dimensionality. Reference model `conv_ref` has one dimension while the reference model `growth_ref` and the main model have two dimensions. Dimensions in the single cell references on reference models are specified using the `CV` function thus relating the cells in main model with the reference model. This specification, in effect, is performing a relational join between main and reference models.

Reference models also help you convert keys to sequence numbers, perform computations using sequence numbers (for example, where a prior period would be used in a subtraction operation), and then convert sequence numbers back to keys. For example, consider a view that assigns sequence numbers to years:

```
CREATE or REPLACE VIEW year_2_seq (i, year) AS
SELECT ROW_NUMBER() OVER (ORDER BY calendar_year), calendar_year
FROM (SELECT DISTINCT calendar_year FROM TIMES);
```

This view can define two lookup tables: integer-to-year `i2y`, which maps sequence numbers to integers, and year-to-integer `y2i`, which performs the reverse mapping. The references `y2i.i[year]` and `y2i.i[year] - 1` return sequence numbers of the current and previous years respectively and the reference `i2y.y[y2i.i[year]-1]` returns the year key value of the previous year. The following query demonstrates such a usage of reference models:

```
SELECT country, product, year, sales, prior_period
FROM sales_view
MODEL
  REFERENCE y2i ON (SELECT year, i FROM year_2_seq) DIMENSION BY (year y)
   MEASURES (i)
  REFERENCE i2y ON (SELECT year, i FROM year_2_seq) DIMENSION BY (i)
   MEASURES (year y)
  MAIN projection2 PARTITION BY (country)
  DIMENSION BY (product, year)
  MEASURES (sales, CAST(NULL AS NUMBER) prior_period)
(prior_period[ANY, ANY] = sales[CV(product), i2y.y[y2i.i[CV(year)]-1]])
ORDER BY country, product, year;
```

Nesting of reference model cell references is evident in the preceding example. Cell reference on the reference model `y2i` is nested inside the cell reference on `i2y` which, in turn, is nested in the cell reference on the main SQL model. There is no limitation on the levels of nesting you can have on reference model cell references. However, you can only have two levels of nesting on the main SQL model cell references.

Finally, the following are restrictions on the specification and usage of reference models:

- Reference models cannot have a `PARTITION BY` clause.
- The query block on which the reference model is defined cannot be correlated to an outer query.

- Reference models must be named and their names should be unique.
- All references to the cells of a reference model should be single cell references.

Advanced Topics in SQL Modeling

This section discusses more advanced topics in SQL modeling, and includes:

- FOR Loops
- Iterative Models
- Rule Dependency in AUTOMATIC ORDER Models
- Ordered Rules
- Analytic Functions
- Unique Dimensions Versus Unique Single References

FOR Loops

The MODEL clause provides a FOR construct that can be used inside rules to express computations more compactly. It can be used on both the left and right side of a rule. FOR loops are treated as positional references when on the left side of a rule. For example, consider the following computation, which estimates the sales of several products for 2004 to be 10% higher than their sales for 2003:

```
RULES UPSERT
(sales['Bounce', 2004] = 1.1 * sales['Bounce', 2003],
 sales['Standard Mouse Pad', 2004] = 1.1 * sales['Standard Mouse Pad', 2003],
...
 sales['Y Box', 2004] = 1.1 * sales['Y Box', 2003])
```

The UPSERT option is used in this computation so that cells for these products and 2004 will be inserted if they are not previously present in the multi-dimensional array. This is rather bulky as you have to have as many rules as there are products. Using the FOR construct, this computation can be represented compactly and with exactly the same semantics as in:

```
RULES UPSERT
(sales[FOR product IN ('Bounce', 'Standard Mouse Pad', ..., 'Y Box'), 2004] =
   1.1 * sales[CV(product), 2003])
```

If you write a specification similar to this, but without the FOR keyword as in the following:

```
RULES UPSERT
(sales[product IN ('Bounce', 'Standard Mouse Pad', ..., 'Y Box'), 2004] =
   1.1 * sales[CV(product), 2003])
```

You would get UPDATE semantics even though you have specified UPSERT. In other words, existing cells will be updated but no new cells will be created by this specification. This is because the multi-cell reference on product is a symbolic reference and symbolic references do not permit insertion of new cells. You can view a FOR construct as a macro that generates multiple rules with positional references from a single rule, thus preserving the UPSERT semantics. Conceptually, the following rule:

```
sales[FOR product IN ('Bounce', 'Standard Mouse Pad', ..., 'Y Box'),
      FOR year IN (2004, 2005)] = 1.1 * sales[CV(product), CV(year)-1]
```

Can be treated as an ordered collection of the following rules:

```
sales['Bounce', 2004] = 1.1 * sales[CV(product), CV(year)-1],
sales['Bounce', 2005] = 1.1 * sales[CV(product), CV(year)-1],
sales['Standard Mouse Pad', 2004] = 1.1 *
  sales[CV(product), CV(year)-1],
sales['Standard Mouse Pad', 2005] = 1.1 * sales[CV(product),
 CV(year)-1],
...
sales['Y Box', 2004] = 1.1 * sales[CV(product), CV(year)-1],
sales['Y Box', 2005] = 1.1 * sales[CV(product), CV(year)-1]
```

The FOR construct in the preceding examples is of type FOR dimension IN *(list of values)*. Values in the list should be single-value expressions such as expressions of constants, single-cell references, and so on. In the last example, there are separate FOR constructs on product and year. It is also possible to specify all dimensions using one FOR construct and specify the values using multi-column IN lists. Consider for example, if we want only to estimate sales for Bounce in 2004, Standard Mouse Pad in 2005 and Y Box in 2004 and 2005. This can be formulated as the following:

```
sales[FOR (product, year) IN (('Bounce', 2004), ('Standard Mouse Pad', 2005),
  ('Y Box', 2004), ('Y Box', 2005))] =
     1.1 * sales[CV(product), CV(year)-1]
```

This FOR construct should be of the form FOR (d1, ..., dn) IN ((d1_val1, ..., dn_val1), ..., (d1_valm, ..., dn_valm)] when there are n dimensions d1, ..., dn and m values in the list.

In some cases, the list of values for a dimension in FOR can be retrieved from a table or a subquery. Oracle Database provides a type of FOR construct as in FOR dimension IN *(subquery)* to handle these cases. For example, assume that the products of interest are stored in a table interesting_products, then the following rule estimates their sales in 2004 and 2005:

```
sales[FOR product IN (SELECT product_name FROM interesting_products)
    FOR year IN (2004, 2005)] = 1.1 * sales[CV(product), CV(year)-1]
```

As another example, consider the scenario where you want to introduce a new country, called new_country, with sales that mimic those of Poland for all products and years where there are sales in Poland. This is accomplished by issuing the following statement:

```
SELECT country, product, year, s
FROM sales_view
MODEL
DIMENSION BY (country, product, year)
MEASURES (sales s) IGNORE NAV
RULES UPSERT
(s[FOR (country, product, year) IN
       (SELECT DISTINCT 'new_country', product, year
        FROM sales_view
        WHERE country = 'Poland')] = s['Poland',CV(),CV()])
ORDER BY country, year, product;
```

Note the multi-column IN-list produced by evaluating the subquery in this specification. The subquery used to obtain the IN-list cannot be correlated to outer query blocks.

Note that the upsert list created by the rule is a cross-product of the distinct values for each dimension. For example, if there are 10 values for country, 5 values for year, and 3 values for product, we will generate an upsert list containing 150 cells.

If you know that the values of interest come from a discrete domain, you can use FOR construct FOR dimension FROM value1 TO value2 [INCREMENT | DECREMENT] value3. This specification results in values between value1 and value2 by starting from value1 and incrementing (or decrementing) by value3. The values value1, value2, and value3 should be single-value expressions. For example, the following rule:

```
sales['Bounce', FOR year FROM 2001 TO 2005 INCREMENT 1] =
  sales['Bounce', year=CV(year)-1] * 1.2
```

This is semantically equivalent to the following rules in order:

```
sales['Bounce', 2001] = sales['Bounce', 2000] * 1.2,
sales['Bounce', 2002] = sales['Bounce', 2001] * 1.2,
...
sales['Bounce', 2005] = sales['Bounce', 2004] * 1.2
```

This kind of FOR construct can be used for dimensions of numeric, date and datetime data types. The type for increment/decrement expression value3 should be numeric for numeric dimensions and can be numeric or interval for dimensions of date or datetime types. Also, value3 should be positive. Oracle Database returns an error if you use FOR year FROM 2005 TO 2001 INCREMENT -1. You should use either FOR year FROM 2005 TO 2001 DECREMENT 1 or FOR year FROM 2001 TO 2005 INCREMENT 1.

To generate string values, you can use the FOR construct FOR dimension LIKE string FROM value1 TO value2 [INCREMENT | DECREMENT] value3. The string string should contain only one % character. This specification results in string by replacing % with values between value1 and value2 with appropriate increment/decrement value value3. For example, consider the following rule:

```
sales[FOR product LIKE 'product-%' FROM 1 TO 3 INCREMENT 1, 2003] =
sales[CV(product), 2002] * 1.2
```

This is equivalent to the following:

```
sales['product-1', 2003] = sales['product-1', 2002] * 1.2,
sales['product-2', 2003] = sales['product-2', 2002] * 1.2,
sales['product-3', 2003] = sales['product-3', 2002] * 1.2
```

In SEQUENTIAL ORDER models, rules represented by a FOR construct are evaluated in the order they are generated. On the contrary, rule evaluation order would be dependency based if AUTOMATIC ORDER is specified. For example, the evaluation order for the rules represented by the rule:

```
sales['Bounce', FOR year FROM 2004 TO 2001 DECREMENT 1] =
  1.1 * sales['Bounce', CV(year)-1]
```

For SEQUENTIAL ORDER models, the rules would be generated in this order:

```
sales['Bounce', 2004] = 1.1 * sales['Bounce', 2003],
sales['Bounce', 2003] = 1.1 * sales['Bounce', 2002],
sales['Bounce', 2002] = 1.1 * sales['Bounce', 2001],
sales['Bounce', 2001] = 1.1 * sales['Bounce', 2000]
```

While for AUTOMATIC ORDER models, the order would be equivalent to:

```
sales['Bounce', 2001] = 1.1 * sales['Bounce', 2000],
sales['Bounce', 2002] = 1.1 * sales['Bounce', 2001],
sales['Bounce', 2003] = 1.1 * sales['Bounce', 2002],
sales['Bounce', 2004] = 1.1 * sales['Bounce', 2003]
```

Evaluation of Formulas with FOR Loops

The FOR loop construct provides an iterative mechanism to generate single-value references for a dimension or for all dimensions (in the case of multi-column for IN lists). The evaluation of a formula with FOR loops on its left side basically consists of evaluation of the right side of the formula for each single-value reference generated by these FOR loops and assigning the result to the specified cell with this single-value reference. The generation of these single reference values is called "unfolding the FOR loop". These unfolded cells are evaluated in the order they are generated during the unfolding process.

How unfolding is performed depends on the UPSERT, UPDATE, and UPDATE ALL behavior specified for the rule and the specific characteristics of the rule. To understand this, we need to discuss two stages of query processing: query plan creation and query execution. Query plan creation is a stage where certain rule references are resolved in order to create an efficient query execution plan. Query execution is the stage where all remaining unresolved references must be determined. FOR loops may be unfolded at either query plan generation or at query execution. Below we discuss the details of the unfolding decision.

Unfolding For UPDATE and UPSERT Rules When using UPDATE or UPSERT rules, if unfolding the left side of a rule is guaranteed to generate single cell references, the unfolding is done at query execution. If the unfolding process cannot generate single cell references, unfolding is performed at query plan creation and a copy of the same formula for each generated reference by the unfolding process is created. For example, the unfolding of the following formula occurs at query execution as unfolding generates single cell references:

```
sales[FOR product IN ('prod1', 'prod2'), 2003] = sales[CV(product), 2002] * 1.2
```

However, consider the following formula, where unfolding reference values do not produce single value references due to the existence of a predicate on another dimension:

```
sales[FOR product in ('prod1', 'prod2'), year >= 2003]
   = sales[CV(product), 2002] * 1.2
```

There is no single-value reference on the year dimension, so even when the FOR loop is unfolded on the product dimension, there will be no single-value references on the left side of this formula. This means that the unfolding occurs at query plan creation and physically replace the original formula with the following formulas:

```
sales['prod1', year >= 2003]  = sales[CV(product), 2002] * 1.2,
sales['prod2', year >= 2003]  = sales[CV(product), 2002] * 1.2
```

The analysis and optimizations performed within the MODEL clause are done after unfolding at query plan creation (if that is what occurs), so, from that point on, everything is as if the multiple rules are specified explicitly in the MODEL clause. By performing unfolding at query plan creation in these cases, more accurate analysis and better optimization of formula evaluation is achieved. One thing to note is that there may be an increase in the number of formulas and, if this increase pushes the total number of formulas beyond the maximum limit, Oracle Database signals an error.

Unfolding For UPSERT ALL Rules Rules with UPSERT ALL behavior have a very different approach to unfolding FOR loops. No matter what predicates are used, an UPSERT ALL rule will unfold FOR loops at query execution. This behavior avoids certain FOR loop restrictions discussed in the next section. However, there is a trade-off of fewer restrictions versus more optimized query plans. An UPSERT ALL rule tends toward

slower performance than a similar UPSERT or UPDATE rule, and this should be considered when designing models.

Restrictions on Using FOR Loop Expressions on the Left Side of Formulas Restrictions on the use of FOR loop constructs are determined based on whether the unfolding takes place at query plan creation or at query execution. If a formula with FOR loops on its left side is unfolded at query plan creation (due to the reasons explained in the previous section), the expressions that need to be evaluated for unfolding must be expressions of constants whose values are available at query plan creation. For example, consider the following statement:

```
sales[For product like 'prod%' from ITERATION_NUMBER
to ITERATION_NUMBER+1, year >= 2003] =  sales[CV(product), 2002]*1.2
```

If this rule does not have UPSERT ALL specified for its behavior, it is unfolded at query plan creation. Because the value of the ITERATION_NUMBER is not known at query plan creation, and the value is needed to evaluate start and end expressions, Oracle Database signals an error unless that rule is unfolded at query execution. However, the following rule would be unfolded at query plan creation without any errors: the value of ITERATION_NUMBER is not needed for unfolding in this case, even though it appears as an expression in the FOR loop:

```
sales[For product in ('prod'||ITERATION_NUMBER, 'prod'||(ITERATION_NUMBER+1)),
 year >= 2003] =  sales[CV(product), 2002]*1.2
```

Expressions that have any of the following conditions cannot be evaluated at query plan creation:

- nested cell references
- reference model look-ups
- ITERATION_NUMBER references

Rules with FOR loops that require the results of such expressions causes an error if unfolded at query plan creation. However, these expressions will not cause any error if unfolding is done at query execution.

If a formula has subqueries in its FOR loop constructs and this formula requires compile-time unfolding, these subqueries are evaluated at query plan creation so that unfolding can happen. Evaluating a subquery at query plan creation can render a cursor non-sharable, which means the same query may need to be recompiled every time it is issued. If unfolding of such a formula is deferred to query execution, no compile-time evaluation is necessary and the formula has no impact on the sharability of the cursor.

Subqueries in the FOR loops of a formula can reference tables in the WITH clause if the formula is to be unfolded at query execution. If the formula has to be unfolded at query plan creation, Oracle Database signals an error.

Iterative Models

Using the ITERATE option of the MODEL clause, you can evaluate rules iteratively for a certain number of times, which you can specify as an argument to the ITERATE clause. ITERATE can be specified only for SEQUENTIAL ORDER models and such models are referred to as iterative models. For example, consider the following:

```
SELECT x, s FROM DUAL
MODEL
  DIMENSION BY (1 AS x) MEASURES (1024 AS s)
  RULES UPDATE ITERATE (4)
```

```
(s[1] = s[1]/2);
```

In Oracle, the table DUAL has only one row. Hence this model defines a 1-dimensional array, dimensioned by x with a measure s, with a single element s[1] = 1024. The rule s[1] = s[1]/2 evaluation will be repeated four times. The result of this query is a single row with values 1 and 64 for columns x and s respectively. The number of iterations arguments for the ITERATE clause should be a positive integer constant. Optionally, you can specify an early termination condition to stop rule evaluation before reaching the maximum iteration. This condition is specified in the UNTIL subclause of ITERATE and is checked at the end of an iteration. So, you will have at least one iteration when ITERATE is specified. The syntax of the ITERATE clause is:

```
ITERATE (number_of_iterations) [ UNTIL (condition) ]
```

Iterative evaluation stops either after finishing the specified number of iterations or when the termination condition evaluates to TRUE, whichever comes first.

In some cases, you may want the termination condition to be based on the change, across iterations, in value of a cell. Oracle Database provides a mechanism to specify such conditions in that it enables you to access cell values as they existed before and after the current iteration in the UNTIL condition. Oracle's PREVIOUS function takes a single cell reference as an argument and returns the measure value of the cell as it existed after the previous iteration. You can also access the current iteration number by using the system variable ITERATION_NUMBER, which starts at value 0 and is incremented after each iteration. By using PREVIOUS and ITERATION_NUMBER, you can construct complex termination conditions.

Consider the following iterative model that specifies iteration over rules till the change in the value of s[1] across successive iterations falls below 1, up to a maximum of 1000 times:

```
SELECT x, s, iterations FROM DUAL
MODEL
  DIMENSION BY (1 AS x) MEASURES (1024 AS s, 0 AS iterations)
  RULES ITERATE (1000) UNTIL ABS(PREVIOUS(s[1]) - s[1]) < 1
 (s[1] = s[1]/2, iterations[1] = ITERATION_NUMBER);
```

The absolute value function (ABS) can be helpful for termination conditions because you may not know if the most recent value is positive or negative. Rules in this model will be iterated over 11 times as after 11th iteration the value of s[1] would be 0.5. This query results in a single row with values 1, 0.5, 10 for x, s and iterations respectively.

You can use the PREVIOUS function only in the UNTIL condition. However, ITERATION_NUMBER can be anywhere in the main model. In the following example, ITERATION_NUMBER is used in cell references:

```
SELECT country, product, year, sales
FROM sales_view
MODEL
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  IGNORE NAV
  RULES ITERATE(3)
(sales['Bounce', 2002 + ITERATION_NUMBER] = sales['Bounce', 1999
  + ITERATION_NUMBER]);
```

This statement achieves an array copy of sales of Bounce from cells in the array 1999-2001 to 2002-2005.

Rule Dependency in AUTOMATIC ORDER Models

Oracle Database determines the order of evaluation of rules in an AUTOMATIC ORDER model based on their dependencies. A rule is evaluated only after the rules it depends on are evaluated. The algorithm chosen to evaluate the rules is based on the dependency analysis and whether rules in your model have circular (or cyclical) dependencies. A cyclic dependency can be of the form "rule A depends on B and rule B depends on A" or of the self-cyclic "rule depending on itself" form. An example of the former is:

```
sales['Bounce', 2002] = 1.5 * sales['Y Box', 2002],
sales['Y Box', 2002] = 100000 / sales['Bounce', 2002
```

An example of the latter is:

```
sales['Bounce', 2002] = 25000 / sales['Bounce', 2002]
```

However, there is no self-cycle in the following rule as different measures are being accessed on the left and right side:

```
projected_sales['Bounce', 2002] = 25000 / sales['Bounce', 2002]
```

When the analysis of an AUTOMATIC ORDER model finds that the rules have no circular dependencies, Oracle Database evaluates the rules in their dependency order. For example, in the following AUTOMATIC ORDER model:

```
MODEL DIMENSION BY (prod, year) MEASURES (sale sales) IGNORE NAV
  RULES AUTOMATIC ORDER
 (sales['SUV', 2001] = 10000,
  sales['Standard Mouse Pad', 2001] = sales['Finding Fido', 2001]
    * 0.10 + sales['Boat', 2001] * 0.50,
  sales['Boat', 2001] = sales['Finding Fido', 2001]
    * 0.25 + sales['SUV', 2001]* 0.75,
  sales['Finding Fido', 2001] = 20000)
```

Rule 2 depends on rules 3 and 4, while rule 3 depends on rules 1 and 4, and rules 1 and 4 do not depend on any rule. Oracle, in this case, will find that the rule dependencies are acyclic and evaluate rules in one of the possible evaluation orders (1, 4, 3, 2) or (4, 1, 3, 2). This type of rule evaluation is called an ACYCLIC algorithm.

In some cases, Oracle Database may not be able to ascertain that your model is acyclic even though there is no cyclical dependency among the rules. This can happen if you have complex expressions in your cell references. Oracle Database assumes that the rules are cyclic and employs a CYCLIC algorithm that evaluates the model iteratively based on the rules and data. Iteration stops as soon as convergence is reached and the results are returned. Convergence is defined as the state in which further executions of the model will not change values of any of the cell in the model. Convergence is certain to be reached when there are no cyclical dependencies.

If your AUTOMATIC ORDER model has rules with cyclical dependencies, Oracle Database employs the earlier mentioned CYCLIC algorithm. Results are produced if convergence can be reached within the number of iterations Oracle is going to try the algorithm. Otherwise, Oracle reports a cycle detection error. You can circumvent this problem by manually ordering the rules and specifying SEQUENTIAL ORDER.

Ordered Rules

An ordered rule is one that has ORDER BY specified on the left side. It accesses cells in the order prescribed by ORDER BY and applies the right side computation. When you have ANY or symbolic references on the left side of a rule but without the ORDER BY

clause, Oracle might return an error saying that the rule's results depend on the order in which cells are accessed and hence are non-deterministic. Consider the following SEQUENTIAL ORDER model:

```
SELECT t, s
FROM sales, times
WHERE sales.time_id = times.time_id
GROUP BY calendar_year
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES SEQUENTIAL ORDER
  (s[ANY] = s[CV(t)-1]);
```

This query attempts to set, for all years t, sales s value for a year to the sales value of the prior year. Unfortunately, the result of this rule depends on the order in which the cells are accessed. If cells are accessed in the ascending order of year, the result would be that of column 3 in Table 23–1. If they are accessed in descending order, the result would be that of column 4.

Table 23–1 Ordered Rules

t	s	If ascending	If descending
1998	1210000982	null	null
1999	1473757581	null	1210000982
2000	2376222384	null	1473757581
2001	1267107764	null	2376222384

If you want the cells to be considered in descending order and get the result given in column 4, you should specify:

```
SELECT t, s
FROM sales, times
WHERE sales.time_id = times.time_id
GROUP BY calendar_year
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES SEQUENTIAL ORDER
  (s[ANY] ORDER BY t DESC = s[CV(t)-1]);
```

In general, you can use any ORDER BY specification as long as it produces a unique order among cells that match the left side cell reference. Expressions in the ORDER BY of a rule can involve constants, measures and dimension keys and you can specify the ordering options [ASC | DESC] [NULLS FIRST | NULLS LAST] to get the order you want.

You can also specify ORDER BY for rules in an AUTOMATIC ORDER model to make Oracle consider cells in a particular order during rule evaluation. Rules are never considered self-cyclic if they have ORDER BY. For example, to make the following AUTOMATIC ORDER model with a self-cyclic formula acyclic:

```
MODEL
  DIMENSION BY (calendar_year t) MEASURES (SUM(amount_sold) s)
  RULES AUTOMATIC ORDER
  (s[ANY] = s[CV(t)-1])
```

You must provide the order in which cells need to be accessed for evaluation using ORDER BY. For example, you can say:

```
s[ANY] ORDER BY t = s[CV(t) - 1]
```

Then Oracle picks an ACYCLIC algorithm (which is certain to produce the result) for formula evaluation.

Analytic Functions

Analytic functions (also known as window functions) can be used in the right side of rules. The ability to use analytic functions adds expressive power and flexibility to the MODEL clause.

The following example combines an analytic function with the MODEL clause. First, we create a view sales_rollup_time that uses the GROUPING_ID function to calculate an identifier for different levels of aggregations. We then use the view in a query that calculates the cumulative sum of sales at both the quarter and year levels.

```
CREATE OR REPLACE VIEW sales_rollup_time
AS
SELECT country_name country, calendar_year year, calendar_quarter_desc quarter,
GROUPING_ID(calendar_year, calendar_quarter_desc) gid, SUM(amount_sold) sale,
COUNT(amount_sold) cnt
FROM sales, times, customers, countries
WHERE sales.time_id = times.time_id AND sales.cust_id = customers.cust_id
  AND customers.country_id = countries.country_id
GROUP BY country_name, calendar_year, ROLLUP(calendar_quarter_desc)
ORDER BY gid, country, year, quarter;

SELECT country, year, quarter, sale, csum
FROM sales_rollup_time
WHERE country IN ('United States of America', 'United Kingdom')
MODEL DIMENSION BY (country, year, quarter)
MEASURES (sale, gid, 0 csum)
(
csum[any, any, any] =
  SUM(sale) OVER (PARTITION BY country, DECODE(gid,0,year,null)
ORDER BY year, quarter
ROWS UNBOUNDED PRECEDING)
)
ORDER BY country, gid, year, quarter;

COUNTRY                          YEAR QUARTER       SALE       CSUM
------------------------------ ---------- ------- ---------- ----------
United Kingdom                   1998 1998-01  484733.96  484733.96
United Kingdom                   1998 1998-02  386899.15  871633.11
United Kingdom                   1998 1998-03  402296.49  1273929.6
United Kingdom                   1998 1998-04  384747.94 1658677.54
United Kingdom                   1999 1999-01  394911.91  394911.91
United Kingdom                   1999 1999-02  331068.38  725980.29
United Kingdom                   1999 1999-03  383982.61  1109962.9
United Kingdom                   1999 1999-04  398147.59 1508110.49
United Kingdom                   2000 2000-01  424771.96  424771.96
United Kingdom                   2000 2000-02  351400.62  776172.58
United Kingdom                   2000 2000-03  385137.68 1161310.26
United Kingdom                   2000 2000-04   390912.8 1552223.06
United Kingdom                   2001 2001-01  343468.77  343468.77
United Kingdom                   2001 2001-02  415168.32  758637.09
United Kingdom                   2001 2001-03  478237.29 1236874.38
United Kingdom                   2001 2001-04  437877.47 1674751.85
United Kingdom                   1998         1658677.54 1658677.54
United Kingdom                   1999         1508110.49 3166788.03
```

```
United Kingdom                          2000         1552223.06 4719011.09
United Kingdom                          2001         1674751.85 6393762.94
...  /*and similar output for the US*/
```

There are some specific restrictions when using analytic functions. See "Rules and Restrictions when Using SQL for Modeling" on page 23-33 for more information.

Unique Dimensions Versus Unique Single References

The MODEL clause, in its default behavior, requires the PARTITION BY and DIMENSION BY keys to uniquely identify each row in the input to the model. Oracle verifies that and returns an error if the data is not unique. Uniqueness of the input rowset on the PARTITION BY and DIMENSION BY keys guarantees that any single cell reference accesses one and only one cell in the model. You can specify an optional UNIQUE DIMENSION keyword in the MODEL clause to make this behavior explicit. For example, the following query:

```
SELECT country, product, sales
FROM sales_view
WHERE country IN ('France', 'Poland')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product) MEASURES (sales sales)
  IGNORE NAV RULES UPSERT
(sales['Bounce'] = sales['All Products'] * 0.24);
```

This would return a uniqueness violation error as the rowset input to model is not unique on country and product because year is also needed:

```
ERROR at line 2:
ORA-32638: Non unique addressing in MODEL dimensions
```

However, the following query does not return such an error:

```
SELECT country, product, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (product, year) MEASURES (sales sales)
  RULES UPSERT
(sales['Bounce', 2003] = sales['All Products', 2002] * 0.24);
```

Input to the MODEL clause in this case is unique on country, product, and year as shown in:

```
COUNTRY   PRODUCT                          YEAR   SALES
-------   -------------------------------  ----   --------
Italy     1.44MB External 3.5" Diskette    1998    3141.84
Italy     1.44MB External 3.5" Diskette    1999    3086.87
Italy     1.44MB External 3.5" Diskette    2000    3440.37
Italy     1.44MB External 3.5" Diskette    2001     855.23
...
```

If you want to relax this uniqueness checking, you can specify UNIQUE SINGLE REFERENCE keyword. This can save processing time. In this case, the MODEL clause checks the uniqueness of only the single cell references appearing on the right side of rules. So the query that returned the uniqueness violation error would be successful if you specify UNIQUE SINGLE REFERENCE instead of UNIQUE DIMENSION.

Another difference between UNIQUE DIMENSION and UNIQUE SINGLE REFERENCE semantics is the number of cells that can be updated by a rule with a single cell

reference on left side. In the case of UNIQUE DIMENSION, such a rule can update at most one row as only one cell would match the single cell reference on the left side. This is because the input rowset would be unique on PARTITION BY and DIMENSION BY keys. With UNIQUE SINGLE REFERENCE, all cells that match the left side single cell reference would be updated by the rule.

Rules and Restrictions when Using SQL for Modeling

The following general rules and restrictions apply when using the MODEL clause:

- The only columns that can be updated are the columns specified in the MEASURES subclause of the main SQL model. Measures of reference models cannot be updated.
- The MODEL clause is evaluated after all clauses in the query block except SELECT DISTINCT, and ORDER BY clause are evaluated. These clauses and expressions in the SELECT list are evaluated after the MODEL clause.
- If your query has a MODEL clause, then the query's SELECT and ORDER BY lists cannot contain aggregates or analytic functions. If needed, these can be specified in PARTITION BY, DIMENSION BY, and MEASURES lists and need to be aliased. Aliases can then be used in the SELECT or ORDER BY clauses. In the following example, the analytic function RANK is specified and aliased in the MEASURES list of the MODEL clause, and its alias is used in the SELECT list so that the outer query can order resulting rows based on their ranks.

```
SELECT country, product, year, s, RNK
FROM (SELECT country, product, year, s, rnk
      FROM sales_view
      MODEL
        PARTITION BY (country) DIMENSION BY (product, year)
        MEASURES (sales s, year y, RANK() OVER (ORDER BY sales) rnk)
        RULES UPSERT
          (s['Bounce Increase 90-99', 2001] =
             REGR_SLOPE(s, y) ['Bounce', year BETWEEN 1990 AND 2000],
           s['Bounce', 2001] = s['Bounce', 2000] *
          (1+s['Bounce increase 90-99', 2001])))
WHERE product <> 'Bounce Increase 90-99'
ORDER BY country, year, rnk, product;
```

- When there is a multi-cell reference on the right hand side of a rule, you need to apply a function to aggregate the measure values of multiple cells referenced into a single value. You can use any kind of aggregate function for this purpose: regular, analytic aggregate (inverse percentile, hypothetical rank and distribution), or user-defined aggregate.
- Only rules with positional single cell references on the left side have UPSERT semantics. All other rules have UPDATE semantics, even when you specify the UPSERT option for them.
- Negative increments are not allowed in FOR loops. Also, no empty FOR loops are allowed. FOR d FROM 2005 TO 2001 INCREMENT -1 is illegal. You should use FOR d FROM 2005 TO 2001 DECREMENT 1 instead. FOR d FROM 2005 TO 2001 INCREMENT 1 is illegal as it designates an empty loop.
- You cannot use nested query expressions (subqueries) in rules except in the FOR construct. For example, it would be illegal to issue the following:

```
SELECT *
FROM sales_view WHERE country = 'Poland'
MODEL DIMENSION BY (product, year)
```

```
  MEASURES (sales sales)
  RULES UPSERT
  (sales['Bounce', 2003] = sales['Bounce', 2002] +
   (SELECT SUM(sales) FROM sales_view));
```

This is because the rule has a subquery on its right side. Instead, you can rewrite the preceding query in the following legal way:

```
SELECT *
FROM sales_view WHERE country = 'Poland'
MODEL DIMENSION BY (product, year)
  MEASURES (sales sales, (SELECT SUM(sales) FROM sales_view) AS grand_total)
  RULES UPSERT
  (sales['Bounce', 2003] =sales['Bounce', 2002] +
   grand_total['Bounce', 2002]);
```

- You can also use subqueries in the `FOR` construct specified on the left side of a rule. However, they:
  - Cannot be correlated
  - Must return fewer than 10,000 rows
  - Cannot be a query defined in the `WITH` clause
  - Will make the cursor unsharable

Nested cell references have the following restrictions:

- Nested cell references must be single cell references. Aggregates on nested cell references are not supported. So, it would be illegal to say `s['Bounce', MAX(best_year)['Bounce', ANY]]`.
- Only one level of nesting is supported for nested cell references on the main model. So, for example, `s['Bounce', best_year['Bounce', 2001]]` is legal, but `s['Bounce', best_year['Bounce', best_year['Bounce', 2001]]]` is not.
- Nested cell references appearing on the left side of rules in an `AUTOMATIC ORDER` model should not be updated in any rule of the model. This restriction ensures that the rule dependency relationships do not arbitrarily change (and hence cause non-deterministic results) due to updates to reference measures.

  There is no such restriction on nested cell references in a `SEQUENTIAL ORDER` model. Also, this restriction is not applicable on nested references appearing on the right side of rules in both `SEQUENTIAL` or `AUTOMATIC ORDER` models.

Reference models have the following restrictions:

- The query defining the reference model cannot be correlated to any outer query. It can, however, be a query with subqueries, views, and so on.
- Reference models cannot have a `PARTITION BY` clause.
- Reference models cannot be updated.

Window functions have the following restrictions:

- The expressions in the `OVER` clause can be expressions of constants, measures, keys from `PARTITION BY` and `DIMENSION BY` of the `MODEL` clause, and single cell expressions. Aggregates are not permitted inside the `OVER` clause. Therefore, the following is okay:

```
rnk[ANY, ANY, ANY] = RANK() (PARTITION BY prod, country ORDER BY sale)
```

While the following is not:

```
rnk[ANY, ANY, ANY] = RANK() (PARTITION BY prod, country ORDER BY SUM(sale))
```

- Rules with window functions on their right side cannot have an ORDER BY clause on their left side.
- Window functions and aggregate functions cannot both be on the right side of a rule.
- Window functions can only be used on the right side of an UPDATE rule.
- If a rule has a FOR loop on its left side, a window function cannot be used on the right side of the rule.

Performance Considerations with SQL Modeling

The following sections describe topics that affect performance when using the MODEL clause:

- Parallel Execution
- Aggregate Computation
- Using EXPLAIN PLAN to Understand Model Queries

Parallel Execution

MODEL clause computation is scalable in terms of the number of processors you have. Scalability is achieved by performing the MODEL computation in parallel across the partitions defined by the PARTITION BY clause. Data is distributed among processing elements (also called parallel query slaves) based on the PARTITION BY key values such that all rows with the same values for the PARTITION BY keys will go to the same slave. Note that the internal processing of partitions will not create a one-to-one match of logical and internally processed partitions. This way, each slave can finish MODEL clause computation independent of other slaves. The data partitioning can be hash based or range based. Consider the following MODEL clause:

```
MODEL
  PARTITION BY (country) DIMENSION BY (product, time) MEASURES (sales)
  RULES UPDATE
  (sales['Bounce', 2002] = 1.2 * sales['Bounce', 2001],
   sales['Car', 2002] = 0.8 * sales['Car', 2001])
```

Here input data will be partitioned among slaves based on the PARTITION BY key country and this partitioning can be hash or range based. Each slave will evaluate the rules on the data it receives.

Parallelism of the model computation is governed or limited by the way you specify the MODEL clause. If your MODEL clause has no PARTITION BY keys, then the computation cannot be parallelized (with exceptions mentioned in the following). If PARTITION BY keys have very low cardinality, then the degree of parallelism will be limited. In such cases, Oracle identifies the DIMENSION BY keys that can used for partitioning. For example, consider a MODEL clause equivalent to the preceding one, but without PARTITION BY keys as in the following:

```
MODEL
  DIMENSION BY (country, product, time) MEASURES (sales)
  RULES UPDATE
  (sales[ANY, 'Bounce', 2002] = 1.2 * sales[CV(country), 'Bounce', 2001],
   sales[ANY, 'Car', 2002] = 0.8 * sales[CV(country), 'Car', 2001])
```

In this case, Oracle Database identifies that it can use the DIMENSION BY key country for partitioning and uses region as the basis of internal partitioning. It partitions the data among slaves on country and thus effects parallel execution.

Aggregate Computation

The MODEL clause processes aggregates in two different ways: first, the regular fashion in which data in the partition is scanned and aggregated and second, an efficient window style aggregation. The first type as illustrated in the following introduces a new dimension member ALL_2002_products and computes its value to be the sum of year 2002 sales for all products:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time) MEASURES (sale sales)
RULES UPSERT
 (sales['ALL_2002_products', 2002] = SUM(sales)[ANY, 2002])
```

To evaluate the aggregate sum in this case, each partition will be scanned to find the cells for 2002 for all products and they will be aggregated. If the left side of the rule were to reference multiple cells, then Oracle will have to compute the right side aggregate by scanning the partition for each cell referenced on the left. For example, consider the following example:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time)
  MEASURES (sale sales, 0 avg_exclusive)
  RULES UPDATE
  (avg_exclusive[ANY, 2002] =  AVG(sales)[product <> CV(product), CV(time)])
```

This rule calculates a measure called avg_exclusive for every product in 2002. The measure avg_exclusive is defined as the average sales of all products excluding the current product. In this case, Oracle scans the data in a partition for every product in 2002 to calculate the aggregate, and this may be expensive.

Oracle Database optimizes the evaluation of such aggregates in some scenarios with window-style computation as used in analytic functions. These scenarios involve rules with multi-cell references on their left side and computing window computations such as moving averages, cumulative sums and so on. Consider the following example:

```
MODEL PARTITION BY (country) DIMENSION BY (product, time)
  MEASURES (sale sales, 0 mavg)
  RULES UPDATE
  (mavg[product IN ('Bounce', 'Y Box', 'Mouse Pad'), ANY] =
   AVG(sales)[CV(product), time BETWEEN CV(time)
   AND CV(time) - 2])
```

It computes the moving average of sales for products Bounce, Y Box, and Mouse Pad over a three year period. It would be very inefficient to evaluate the aggregate by scanning the partition for every cell referenced on the left side. Oracle identifies the computation as being in window-style and evaluates it efficiently. It sorts the input on product, time and then scans the data once to compute the moving average. You can view this rule as an analytic function being applied on the sales data for products Bounce, Y Box, and Mouse Pad:

```
AVG(sales) OVER (PARTITION BY product ORDER BY time
RANGE BETWEEN 2 PRECEDING AND CURRENT ROW)
```

This computation style is called WINDOW (IN MODEL) SORT. This style of aggregation is applicable when the rule has a multi-cell reference on its left side with no ORDER BY, has a simple aggregate (SUM, COUNT, MIN, MAX, STDEV, and VAR) on its

right side, only one dimension on the right side has a boolean predicate (<, <=, >, >=, BETWEEN), and all other dimensions on the right are qualified with CV.

Using EXPLAIN PLAN to Understand Model Queries

Oracle's explain plan facility is fully aware of models. You will see a line in your query's main explain plan output showing the model and the algorithm used. Reference models are tagged with the keyword REFERENCE in the plan output. Also, Oracle annotates the plan with WINDOW (IN MODEL) SORT if any of the rules qualify for window-style aggregate computation.

By examining an explain plan, you can find out the algorithm chosen to evaluate your model. If your model has SEQUENTIAL ORDER semantics, then ORDERED is displayed. For AUTOMATIC ORDER models, Oracle displays ACYCLIC or CYCLIC based on whether it chooses ACYCLIC or CYCLIC algorithm for evaluation. In addition, the plan output will have an annotation FAST in case of ORDERED and ACYCLIC algorithms if all left side cell references are single cell references and aggregates, if any, on the right side of rules are simple arithmetic non-distinct aggregates like SUM, COUNT, AVG, and so on. Rule evaluation in this case would be highly efficient and hence the annotation FAST. Thus, the output you will see in the explain plan would be MODEL {ORDERED [FAST] | ACYCLIC [FAST] | CYCLIC}.

Using ORDERED FAST: Example

This model has only single cell references on the left side of rules and the aggregate AVG on the right side of first rule is a simple non-distinct aggregate:

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT
  (sales['Bounce', 2003] = AVG(sales)[ANY, 2002] * 1.24,
   sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25);
```

Using ORDERED: Example

Because the left side of the second rule is a multi-cell reference, the FAST method will not be chosen in the following:

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT
  (sales['Bounce', 2003] = AVG(sales)[ANY, 2002] * 1.24,
   sales[prod <> 'Bounce', 2003] = sales['Bounce', 2003] * 0.25);
```

Using ACYCLIC FAST: Example

Rules in this model are not cyclic and the explain plan will show ACYCLIC. The FAST method is chosen in this case as well.

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
```

```
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT AUTOMATIC ORDER
  (sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25,
   sales['Bounce', 2003] = sales['Bounce', 2002] / SUM(sales)[ANY, 2002] * 2 *
   sales['All Products', 2003],
   sales['All Products', 2003] = 200000);
```

Using ACYCLIC: Example

Rules in this model are not cyclic. The `PERCENTILE_DISC` aggregate that gives the median sales for year 2002, in the second rule is not a simple aggregate function. Therefore, Oracle will not choose the `FAST` method, and the explain plan will just show `ACYCLIC`.

```
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  RULES UPSERT AUTOMATIC ORDER
  (sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25,
   sales['Bounce',2003] = PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY
   sales)[ANY,2002] / SUM(sales)[ANY, 2002] * 2 * sales['All Products', 2003],
   sales['All Products', 2003] = 200000);
```

Using CYCLIC: Example

Oracle chooses `CYCLIC` algorithm for this model as there is a cycle among second and third rules.

```
EXPLAIN PLAN FOR
SELECT country, prod, year, sales
FROM sales_view
WHERE country IN ('Italy', 'Japan')
MODEL UNIQUE DIMENSION
  PARTITION BY (country) DIMENSION BY (prod, year) MEASURES (sale sales)
  IGNORE NAV RULES UPSERT AUTOMATIC ORDER
  (sales['All Products', 2003] = 200000,
   sales['Y Box', 2003] = sales['Bounce', 2003] * 0.25,
   sales['Bounce', 2003] = sales['Y Box', 2003] +
    (sales['Bounce', 2002] / SUM(sales)[ANY, 2002] * 2 *
     sales['All Products', 2003]));
```

Examples of SQL Modeling

The examples in this section assume that in addition to `sales_view`, you have the following view defined. It finds monthly totals of sales and quantities by product and country.

```
CREATE VIEW sales_view2 AS
SELECT country_name country, prod_name product, calendar_year year,
  calendar_month_name month, SUM(amount_sold) sale, COUNT(amount_sold) cnt
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
      customers.country_id = countries.country_id
GROUP BY country_name, prod_name, calendar_year, calendar_month_name;
```

Example 1 Calculating Sales Differences

Show the sales for Italy and Spain and the difference between the two for each product. The difference should be placed in a new row with `country = 'Diff Italy-Spain'`.

```
SELECT product, country, sales
FROM sales_view
WHERE country IN ('Italy', 'Spain')
GROUP BY product, country
MODEL
  PARTITION BY (product) DIMENSION BY (country) MEASURES (SUM(sales) AS sales)
  RULES UPSERT
  (sales['DIFF ITALY-SPAIN'] = sales['Italy'] - sales['Spain']);
```

Example 2 Calculating Percentage Change

If sales for each product in each country grew (or declined) at the same monthly rate from November 2000 to December 2000 as they did from October 2000 to November 2000, what would the fourth quarter's sales be for the whole company and for each country?

```
SELECT country, SUM(sales)
FROM (SELECT product, country, month, sales
      FROM sales_view2
      WHERE year=2000 AND month IN ('October','November')
MODEL
  PARTITION BY (product, country) DIMENSION BY (month) MEASURES (sale sales)
  RULES
   (sales['December']=(sales['November'] /sales['October']) *sales['November']))
GROUP BY GROUPING SETS ((),(country));
```

Example 3 Calculating Net Present Value

You want to calculate the net present value (NPV) of a series of periodic cash flows. Your scenario involves two projects, each of which starts with an initial investment at time 0, represented as a negative cash flow. The initial investment is followed by three years of positive cash flow. First, create a table (`cash_flow`) and populate it with some data, as in the following statements:

```
CREATE TABLE cash_flow (year DATE, i INTEGER, prod VARCHAR2(3), amount NUMBER);
INSERT INTO cash_flow VALUES (TO_DATE('1999', 'YYYY'),  0,  'vcr',  -100.00);
INSERT INTO cash_flow VALUES (TO_DATE('2000', 'YYYY'),  1,  'vcr',   12.00);
INSERT INTO cash_flow VALUES (TO_DATE('2001', 'YYYY'),  2,  'vcr',  10.00);
INSERT INTO cash_flow VALUES (TO_DATE('2002', 'YYYY'),  3,  'vcr',  20.00);
INSERT INTO cash_flow VALUES (TO_DATE('1999', 'YYYY'),  0,  'dvd',  -200.00);
INSERT INTO cash_flow VALUES (TO_DATE('2000', 'YYYY'),  1,  'dvd',  22.00);
INSERT INTO cash_flow VALUES (TO_DATE('2001', 'YYYY'),  2,  'dvd',  12.00);
INSERT INTO cash_flow VALUES (TO_DATE('2002', 'YYYY'),  3,  'dvd',  14.00);
```

To calculate the NPV using a discount rate of 0.14, issue the following statement:

```
SELECT year, i, prod, amount, npv
FROM cash_flow
MODEL PARTITION BY (prod)
  DIMENSION BY (i)
  MEASURES (amount, 0 npv, year)
  RULES
    (npv[0] = amount[0],
     npv[i !=0] ORDER BY i =
       amount[CV()]/ POWER(1.14,CV(i)) + npv[CV(i)-1]);

YEAR               I PRO     AMOUNT        NPV
```

```
--------- ---------- --- ---------- ----------
01-AUG-99          0 dvd       -200       -200
01-AUG-00          1 dvd         22 -180.70175
01-AUG-01          2 dvd         12 -171.46814
01-AUG-02          3 dvd         14 -162.01854
01-AUG-99          0 vcr       -100       -100
01-AUG-00          1 vcr         12 -89.473684
01-AUG-01          2 vcr         10 -81.779009
01-AUG-02          3 vcr         20 -68.279579
```

Example 4 Calculating Using Simultaneous Equations

You want your interest expenses to equal 30% of your net income (net=pay minus tax minus interest). Interest is tax deductible from gross, and taxes are 38% of salary and 28% capital gains. You have salary of $100,000 and capital gains of $15,000. Net income, taxes, and interest expenses are unknown. Observe that this is a simultaneous equation (net depends on interest, which depends on net), thus the `ITERATE` clause is included.

First, create a table called `ledger`:

```
CREATE TABLE  ledger  (account  VARCHAR2(20), balance  NUMBER(10,2) );
```

Then, insert the following five rows:

```
INSERT INTO ledger VALUES  ('Salary', 100000);
INSERT INTO ledger VALUES  ('Capital_gains', 15000);
INSERT INTO ledger VALUES  ('Net', 0);
INSERT INTO ledger VALUES  ('Tax', 0);
INSERT INTO ledger VALUES  ('Interest', 0);
```

Next, issue the following statement:

```
SELECT s, account
FROM ledger
MODEL
  DIMENSION  BY (account) MEASURES (balance s)
  RULES  ITERATE (100)
  (s['Net']=s['Salary']-s['Interest']-s['Tax'],
 s['Tax']=(s['Salary']-s['Interest'])*0.38 + s['Capital_gains']*0.28,
  s['Interest']=s['Net']*0.30);
```

The output (with numbers rounded) is:

```
         S ACCOUNT
---------- --------------------
    100000 Salary
     15000 Capital_gains
48735.2445 Net
36644.1821 Tax
14620.5734 Interest
```

Example 5 Calculating Using Regression

The sales of Bounce in 2001 will increase in comparison to 2000 as they did in the last three years (between 1998 and 2000). To calculate the increase, use the regression function `REGR_SLOPE` as follows. Because we are calculating the next period's value, it is sufficient to add the slope to the 2000 value.

```
SELECT * FROM
 (SELECT country, product, year, projected_sale, sales
  FROM sales_view
  WHERE country IN ('Italy', 'Japan') AND product IN ('Bounce')
```

```
MODEL
  PARTITION BY (country) DIMENSION BY (product, year)
  MEASURES (sales sales, year y, CAST(NULL AS NUMBER) projected_sale) IGNORE NAV
  RULES UPSERT
  (projected_sale[FOR product IN ('Bounce'), 2001] =
             sales[CV(), 2000] +
             REGR_SLOPE(sales, y)[CV(), year BETWEEN 1998 AND 2000]))
ORDER BY country, product, year;
```

The output is as follows:

```
COUNTRY    PRODUCT           YEAR   PROJECTED_SALE  SALES
-------    -------           ----   --------------  -------
Italy      Bounce            1999                    2474.78
Italy      Bounce            2000                    4333.69
Italy      Bounce            2001           6192.6   4846.3
Japan      Bounce            1999                    2961.3
Japan      Bounce            2000                    5133.53
Japan      Bounce            2001          7305.76   6303.6
```

Example 6 Calculating Mortgage Amortization

This example creates mortgage amortization tables for any number of customers, using information about mortgage loans selected from a table of mortgage facts. First, create two tables and insert needed data:

- `mortgage_facts`

  Holds information about individual customer loans, including the name of the customer, the fact about the loan that is stored in that row, and the value of that fact. The facts stored for this example are loan (`Loan`), annual interest rate (`Annual_Interest`), and number of payments (`Payments`) for the loan. Also, the values for two customers, Smith and Jones, are inserted.

  ```
  CREATE TABLE mortgage_facts (customer VARCHAR2(20), fact VARCHAR2(20),
    amount  NUMBER(10,2));
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Loan', 100000);
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Annual_Interest', 12);
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Payments', 360);
  INSERT INTO mortgage_facts  VALUES ('Smith', 'Payment', 0);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Loan', 200000);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Annual_Interest', 12);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Payments', 180);
  INSERT INTO mortgage_facts  VALUES ('Jones', 'Payment', 0);
  ```

- `mortgage`

  Holds output information for the calculations. The columns are customer, payment number (`pmt_num`), principal applied in that payment (`principalp`), interest applied in that payment (`interestp`), and remaining loan balance (`mort_balance`). In order to upsert new cells into a partition, you need to have at least one row pre-existing per partition. Therefore, we seed the mortgage table with the values for the two customers before they have made any payments. This seed information could be easily generated using a SQL `INSERT` statement based on the `mortgage_facts` table.

```
CREATE TABLE mortgage_facts (customer VARCHAR2(20), fact VARCHAR2(20),
   amount  NUMBER(10,2));

INSERT INTO mortgage_facts  VALUES ('Smith', 'Loan', 100000);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payments', 360);
```

```
INSERT INTO mortgage_facts  VALUES ('Smith', 'Payment', 0);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Loan', 200000);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Annual_Interest', 12);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payments', 180);
INSERT INTO mortgage_facts  VALUES ('Jones', 'Payment', 0);

CREATE TABLE mortgage (customer VARCHAR2(20), pmt_num NUMBER(4),
   principalp NUMBER(10,2), interestp NUMBER(10,2), mort_balance NUMBER(10,2));

INSERT INTO mortgage VALUES ('Jones',0, 0, 0, 200000);
INSERT INTO mortgage VALUES ('Smith',0, 0, 0, 100000);
```

The following SQL statement is complex, so individual lines have been annotated as needed. These lines are explained in more detail later.

```
SELECT c, p, m, pp, ip
FROM MORTGAGE
MODEL                                                                  --See 1
REFERENCE R ON
   (SELECT customer, fact, amt                                         --See 2
    FROM mortgage_facts
    MODEL DIMENSION BY (customer, fact) MEASURES (amount amt)          --See 3
      RULES
       (amt[any, 'PaymentAmt']= (amt[CV(),'Loan']*
          Power(1+ (amt[CV(),'Annual_Interest']/100/12),
                  amt[CV(),'Payments']) *
         (amt[CV(),'Annual_Interest']/100/12)) /
         (Power(1+(amt[CV(),'Annual_Interest']/100/12),
                amt[CV(),'Payments']) - 1)
       )
    )
  DIMENSION BY (customer cust, fact) measures (amt)                     --See 4
MAIN amortization
  PARTITION BY (customer c)                                            --See 5
  DIMENSION BY (0 p)                                                   --See 6
  MEASURES (principalp pp, interestp ip, mort_balance m, customer mc)  --See 7
  RULES
    ITERATE(1000) UNTIL (ITERATION_NUMBER+1 =
r.amt[mc[0],'Payments'])                                               --See 8
    (ip[ITERATION_NUMBER+1] = m[CV()-1] *
       r.amt[mc[0], 'Annual_Interest']/1200,                           --See 9
     pp[ITERATION_NUMBER+1] = r.amt[mc[0], 'PaymentAmt'] - ip[CV()],   --See 10
      m[ITERATION_NUMBER+1] = m[CV()-1] - pp[CV()]                     --See 11
    )
ORDER BY c, p;
```

The following numbers refer to the numbers listed in the example:

1: This is the start of the main model definition.

2 through 4: These lines mark the start and end of the reference model labeled `R`. This model defines a `SELECT` statement that calculates the monthly payment amount for each customer's loan. The `SELECT` statement uses its own `MODEL` clause starting at the line labeled 3 with a single rule that defines the `amt` value based on information from the `mortgage_facts` table. The measure returned by reference model `R` is `amt`, dimensioned by customer name `cust` and fact value `fact` as defined in the line labeled 4.

The reference model is computed once and the values are then used in the main model for computing other calculations. Reference model `R` will return a row for each existing row of `mortgage_facts`, and it will return the newly calculated rows for

each customer where the fact type is `Payment` and the `amt` is the monthly payment amount. If we wish to use a specific amount from the `R` output, we address it with the expression `r.amt[<customer_name>,<fact_name>]`.

5: This is the continuation of the main model definition. We will partition the output by customer, aliased as `c`.

6: The main model is dimensioned with a constant value of 0, aliased as `p`. This represents the payment number of a row.

7: Four measures are defined: `principalp (pp)` is the principal amount applied to the loan in the month, `interestp (ip)` is the interest paid that month, `mort_balance (m)` is the remaining mortgage value after the payment of the loan, and `customer (mc)` is used to support the partitioning.

8: This begins the rules block. It will perform the rule calculations up to 1000 times. Because the calculations are performed once for each month for each customer, the maximum number of months that can be specified for a loan is 1000. Iteration is stopped when the `ITERATION_NUMBER+1` equals the amount of payments derived from reference `R`. Note that the value from reference `R` is the `amt` (amount) measure defined in the reference clause. This reference value is addressed as `r.amt[<customer_name>,<fact>]`. The expression used in the iterate line, `"r.amt[mc[0], 'Payments']"` is resolved to be the amount from reference `R`, where the customer name is the value resolved by `mc[0]`. Since each partition contains only one customer, `mc[0]` can have only one value. Thus `"r.amt[mc[0], 'Payments']"` yields the reference clause's value for the number of payments for the current customer. This means that the rules will be performed as many times as there are payments for that customer.

9 through 11: The first two rules in this block use the same type of `r.amt` reference that was explained in 8. The difference is that the ip rule defines the fact value as `Annual_Interest`. Note that each rule refers to the value of one of the other measures. The expression used on the left side of each rule, `"[ITERATION_NUMBER+1]"` will create a new dimension value, so the measure will be upserted into the result set. Thus the result will include a monthly amortization row for all payments for each customer.

The final line of the example sorts the results by customer and loan payment number.

24

Advanced Business Intelligence Queries

This chapter illustrates techniques for handling advanced business intelligence queries. We hope to enhance your understanding of how different SQL features can be used together to perform demanding analyses. Although the features shown here have been addressed on an individual basis in Chapter 21, "SQL for Aggregation in Data Warehouses", Chapter 22, "SQL for Analysis and Reporting", and Chapter 23, "SQL for Modeling", seeing features one at a time gives only a limited sense of how they can work together. Here we show the analytic power available when the features are combined.

What makes a business intelligence query "advanced"? The label is best applied to multistep queries, often involving dimension hierarchies. In such queries, the final result depends on several sets of retrieved data, multiple calculation steps, and the data retrieved may involve multiple levels of a dimension hierarchy. Prime examples of advanced queries are market share calculations based on multiple conditions and sales projections that require filling gaps in data.

The examples in this chapter illustrate using nested inline views, `CASE` expressions, partition outer join, the `MODEL` and `WITH` clauses, analytic SQL functions, and more. Where relevant to the discussion, query plans will be discussed. This chapter includes:

- Examples of Business Intelligence Queries

Examples of Business Intelligence Queries

The queries in this chapter illustrate various business intelligence tasks. The topics of the queries and the features used in each query are:

- Percent change in market share based on complex multistep conditions. It illustrates nested inline views, `CASE` expression, and analytic SQL functions.
- Sales projection with gaps in data filled in. It illustrates the `MODEL` clause together with partition outer join and the `CASE` expression.
- Customer analysis grouping customers into purchase-size buckets. It illustrates the `WITH` clause (query subfactoring) and the analytic SQL functions `percentile_cont` and `width_bucket`.
- Customer item grouping into itemsets. It illustrates calculating frequent itemsets using `DBMS_FREQUENT_ITEMSET.FI_TRANSACTIONAL` as a table function.

Example 1 Percent change in market share of products in a calculated set

What was the percent change in market share for a grouping of my top 20% of products for the current three-month period versus same period year ago for accounts that grew by more than 20 percent in revenue?

We define market share as a product's share of total sales. We do this because there is no data for competitors in the sh sample schema, so the typical share calculation of product sales and competitors' sales is not possible. The processing required for our share calculation is logically similar to a competitive market share calculation.

Here are the pieces of information we find in the query, in the order we need to find them:

1. Cities whose purchases grew by more than 20% during the specified 3-month period, versus the same 3-month period last year. Note that cities are limited to one country, and sales involving no promotion.
2. Top 20% of the products for the group of cities found in the prior step. That is, find sales by product summed across this customer group, and then select the 20% of products with the best sales.
3. The share of sales for each product found in the prior step. That is, using the products group found in the prior step, find each product's share of sales of all products. Find the shares for the same period a year ago and then calculate the change in share between the two years.

The techniques used in this example are:

- This query is performed using the WITH clause and nested inline views. Each inline view has been given a descriptive alias to show its data element, and comment lines indicate the boundaries of each inline view. Although inline views are powerful, we believe that readability and maintenance are much easier if queries are structured to maximize the use of the WITH clause.
- This query does not use the WITH clause as extensively as it might: some of the nested inline views could have been expressed as separate subclauses of the WITH clause. For instance, in the main query, we use two inline views that return just one value. These are used for the denominator of the share calculations. We could have factored out these items and placed them in the WITH clause for greater readability. For a contrast that does use the WITH clause to its maximum, see the example "Customer analysis by grouping customers into buckets" on page 24-6 regarding customer purchase analysis.
- Note the use of CASE expressions within the arguments to SUM functions. The CASE expressions simplify the SQL by acting as an extra set of data filters after the WHERE clause. They allow us to sum one column of sales for a desired date and another column for a different date.

```
WITH prod_list AS                    --START: Top 20% of products
   ( SELECT prod_id prod_subset, cume_dist_prod
    FROM                             --START: All products Sales for city subset
    ( SELECT s.prod_id, SUM(amount_sold),
        CUME_DIST() OVER (ORDER BY SUM(amount_sold)) cume_dist_prod
      FROM sales s, customers c, channels ch,  products p, times t
      WHERE s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
            s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
            s.cust_id = c.cust_id AND
            s.promo_id = 999 AND
            s.time_id  = t.time_id AND t.calendar_quarter_id = 1776 AND
            c.cust_city_id IN
       (SELECT cust_city_id            --START: Top 20% of cities
       FROM
         (
           SELECT cust_city_id, ((new_cust_sales - old_cust_sales)
                / old_cust_sales ) pct_change, old_cust_sales
           FROM
```

```
          (
             SELECT cust_city_id, new_cust_sales, old_cust_sales
             FROM
             (          --START: Cities AND sales for 1 country in 2 periods
               SELECT cust_city_id,
                 SUM(CASE WHEN t.calendar_quarter_id = 1776
                    THEN amount_sold  ELSE  0  END ) new_cust_sales,
                 SUM(CASE WHEN t.calendar_quarter_id = 1772
                    THEN amount_sold ELSE 0 END) old_cust_sales
               FROM sales s, customers c, channels ch,
                    products p, times t
               WHERE s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
                     s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
                     s.cust_id = c.cust_id AND c.country_id = 52790 AND
                     s.promo_id = 999 AND
                     s.time_id  = t.time_id AND
                    (t.calendar_quarter_id = 1776 OR t.calendar_quarter_id =1772)
               GROUP BY cust_city_id
             ) cust_sales_wzeroes
             WHERE old_cust_sales > 0
          )  cust_sales_woutzeroes
        )         --END: Cities and sales for country in 2 periods
        WHERE old_cust_sales > 0 AND  pct_change >= 0.20)
                                  --END: Top 20% of cities
GROUP BY s.prod_id
)  prod_sales                     --END: All products sales for city subset
    WHERE cume_dist_prod > 0.8    --END: Top 20% products
)
                                  --START: Main query bloc
SELECT  prod_id, ( (new_subset_sales/new_tot_sales)
              -    (old_subset_sales/old_tot_sales)
                 ) *100  share_changes
FROM
(                                 --START: Total sales for country in later period
  SELECT  prod_id,
     SUM(CASE WHEN t.calendar_quarter_id = 1776
                 THEN amount_sold  ELSE  0  END )  new_subset_sales,
        (SELECT SUM(amount_sold) FROM sales s, times t, channels ch,
                customers c, countries co, products p
          WHERE s.time_id  = t.time_id AND t.calendar_quarter_id = 1776 AND
                s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
                s.cust_id = c.cust_id AND
                c.country_id = co.country_id AND co.country_total_id = 52806 AND
                s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
                s.promo_id = 999
        )   new_tot_sales,

                                 --END: Total sales for country in later period
                                 --START: Total sales for country in earlier period
     SUM(CASE WHEN t.calendar_quarter_id = 1772
                 THEN amount_sold  ELSE  0  END)  old_subset_sales,
        (SELECT SUM(amount_sold) FROM sales s, times t, channels ch,
                customers c, countries co, products p
          WHERE s.time_id  = t.time_id AND t.calendar_quarter_id = 1772 AND
                s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
                s.cust_id = c.cust_id AND
                c.country_id = co.country_id AND co.country_total_id = 52806 AND

                s.prod_id = p.prod_id AND p.prod_total_id = 1 AND
                s.promo_id = 999
```

```
       )   old_tot_sales
                                --END: Total sales for country in earlier period
 FROM sales s, customers c, countries co, channels ch, times t
 WHERE s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
       s.cust_id = c.cust_id AND
       c.country_id = co.country_id AND co.country_total_id = 52806 AND
       s.promo_id = 999 AND
       s.time_id  = t.time_id AND
      (t.calendar_quarter_id = 1776 OR t.calendar_quarter_id = 1772)
         AND s.prod_id IN
     (SELECT prod_subset FROM prod_list)
  GROUP BY prod_id);
```

Example 2 Sales projection that fills in missing data

This query projects sales for 2002 based on the sales of 2000 and 2001. It finds the most percentage changes in sales from 2000 to 2001 and then adds that to the sales of 2002. While this is a simple calculation, there is an important thing to consider: many products have months with no sales in 2000 and 2001. We want to fill in blank values with the average sales value for the year (based on the months with actual sales). It converts currency values by country into US dollars. Finally, the query returns just the 2002 projection values.

The techniques used in this example are:

- By predefining all possible rows of data with the cross join ahead of the `MODEL` clause, we reduce the processing required by `MODEL`.
- The `MODEL` clause uses a reference model to perform currency conversion.
- By using the `CV` function extensively, we reduce the total number of rules needed to just three.
- The most interesting expression is found in the last rule, which uses a nested rule to find the currency conversion factor. To supply the country name needed for this expression, we define country as both a dimension `c` in the reference model, and a measure `cc` in the main model.

The way this example proceeds is to begin by creating a reference table of currency conversion factors. The table will hold conversion factors for each month for each country. Note that we use a cross join to specify the rows inserted into the table. For our purposes, we only set the conversion factor for one country, Canada.

```
CREATE TABLE currency (
   country         VARCHAR2(20),
   year            NUMBER,
   month           NUMBER,
   to_us           NUMBER);

INSERT INTO currency
(SELECT distinct
SUBSTR(country_name,1,20), calendar_year, calendar_month_number, 1
FROM countries
CROSS JOIN times t
WHERE calendar_year IN (2000,2001,2002)
);
UPDATE currency set to_us=.74 WHERE country='Canada';
```

Here is the projection query. It starts with a `WITH` clause that has two subclauses. The first subclause finds the monthly sales per product by country for the years 2000, 2001, and 2002. The second subclause finds a list of distinct times at the month level.

```
WITH  prod_sales_mo AS       --Product sales per month for one country
(
SELECT country_name c, prod_id p, calendar_year  y,
   calendar_month_number  m, SUM(amount_sold) s
FROM sales s, customers c, times t, countries cn, promotions p, channels ch
WHERE  s.promo_id = p.promo_id AND p.promo_total_id = 1 AND
       s.channel_id = ch.channel_id AND ch.channel_total_id = 1 AND
       s.cust_id=c.cust_id  AND
       c.country_id=cn.country_id AND country_name='France' AND
       s.time_id=t.time_id  AND t.calendar_year IN  (2000, 2001,2002)
GROUP BY cn.country_name, prod_id, calendar_year, calendar_month_number
),
                    -- Time data used for ensuring that model has all dates
time_summary AS
(
  SELECT DISTINCT calendar_year cal_y, calendar_month_number cal_m
  FROM times
  WHERE  calendar_year IN  (2000, 2001, 2002)
)
                   --START: main query block
SELECT c, p, y, m, s,  nr FROM (
SELECT c, p, y, m, s,  nr
FROM prod_sales_mo s
                   --Use partition outer join to make sure that each combination
                   --of country and product has rows for all month values
  PARTITION BY (s.c, s.p)
     RIGHT OUTER JOIN time_summary ts ON
        (s.m = ts.cal_m
         AND s.y = ts.cal_y
        )
MODEL
  REFERENCE curr_conversion ON
      (SELECT country, year, month, to_us
      FROM currency)
      DIMENSION BY (country, year y,month m) MEASURES (to_us)
                               --START: main model
   PARTITION BY (s.c c)
   DIMENSION BY (s.p p, ts.cal_y y, ts.cal_m m)
   MEASURES (s.s s, CAST(NULL AS NUMBER) nr,
             s.c cc ) --country is used for currency conversion
   RULES (
                      --first rule fills in missing data with average values
      nr[ANY, ANY, ANY]
         = CASE WHEN s[CV(), CV(), CV()] IS NOT NULL
              THEN s[CV(), CV(), CV()]
              ELSE ROUND(AVG(s)[CV(), CV(), m BETWEEN 1 AND 12],2)
           END,
                      --second rule calculates projected values for 2002
      nr[ANY, 2002, ANY] = ROUND(
         ((nr[CV(),2001,CV()] - nr[CV(),2000, CV()])
          / nr[CV(),2000, CV()]) * nr[CV(),2001, CV()]
         + nr[CV(),2001, CV()],2),
                      --third rule converts 2002 projections to US dollars
      nr[ANY,y != 2002,ANY]
         = ROUND(nr[CV(),CV(),CV()]
           * curr_conversion.to_us[ cc[CV(),CV(),CV()], CV(y), CV(m)], 2)
)
ORDER BY c, p, y, m)
WHERE y = '2002'
ORDER BY c, p, y, m;
```

Example 3 Customer analysis by grouping customers into buckets

One important way to understand customers is by studying their purchasing patterns and learning the profitability of each customer. This can help us decide if a customer is worth cultivating and what kind of treatment to give it. Because the `sh` sample schema data set includes many customers, a good way to start a profitability analysis is with a high level view: we will find data for a histogram of customer profitability, dividing profitability into 10 ranges (often called "buckets" for histogram analyses).

For each country at an aggregation level of 1 month, we show:

- The data needed for a 10-bucket equiwidth histogram of customer profitability. That is, show the count of customers falling into each of 10 profitability buckets. This is just 10 rows of results, but it involves significant calculations.

For each profitability bucket, we also show:

- The median count of transactions per customer during the month (treating each day's purchases by 1 customer in 1 channel as a single transaction).
- The median transaction size (in local currency) per customer.
- Products that generated the most and least profit.
- Percent change of median transaction count and median transaction size versus last year.

The techniques used in this example illustrate the following:

- Using the `WITH` clause to clarify a query. By dividing the needed data into logical chunks, each of which is expressed in its own `WITH` subclause, we greatly improve readability and maintenance compared to nested inline views. The thorough use of `WITH` subclauses means that the main `SELECT` clause does not need to perform any calculations on the data it retrieves, again contributing to the readability and maintainability of the query.
- Using two analytic SQL functions, `width_bucket` equiwidth histogram buckets and `percentile_cont` to median transaction size and count.

This query shows us the analytic challenges inherent in data warehouse designs: because the `sh` data does not include entries for every transaction, nor a count of transactions, we are forced to make an assumption. In this query, we will make the minimalist interpretation and assume that all products sold to a single customer through a single channel on a single day are part of the same transaction. This approach inevitably undercounts transactions, because some customers will in fact make multiple purchases through the same channel on the same day.

Note that the query below should not be run until a materialized view is created for the initial query subfactor `cust_prod_mon_profit`. Before creating the materialized view, create two additional indexes. Unless these preparatory steps are taken, the query may require significant time to run.

The two additional indexes needed and the main query are as follows:

```
CREATE BITMAP INDEX costs_chan_bix
      ON costs (channel_id)
      LOCAL NOLOGGING COMPUTE STATISTICS;

CREATE BITMAP INDEX costs_promo_bix
      ON costs (promo_id)
      LOCAL NOLOGGING COMPUTE STATISTICS;

WITH cust_prod_mon_profit AS
-- profit by cust, prod, day, channel, promo
```

```
  (SELECT s.cust_id, s.prod_id, s.time_id,
          s.channel_id, s.promo_id,
          s.quantity_sold*(c.unit_price-c.unit_cost) profit,
          s.amount_sold dol_sold, c.unit_price price, c.unit_cost cost
   FROM sales s, costs c
   WHERE s.prod_id=c.prod_id
     AND s.time_id=c.time_id
     AND s.promo_id=c.promo_id
     AND s.channel_id=c.channel_id
     AND s.cust_id in (SELECT cust_id FROM customers cst
                       WHERE cst.country_id = 52770
     AND s.time_id IN (SELECT time_id FROM times t
                       WHERE t.calendar_month_desc = '2000-12'
   ),
-- Transaction Definition:  All products sold through a single channel to a
-- single cust on a single day are assumed to be sold in 1 transaction.
-- Some products in a transacton
-- may be on promotion
-- A customers  daily transaction amount is the sum of ALL products
-- purchased in the same channel in the same day
cust_daily_trans_amt  AS
(  SELECT cust_id, time_id, channel_id,  SUM(dol_sold) cust_daily_trans_amt
   FROM  cust_prod_mon_profit
   GROUP BY cust_id, time_id, channel_id
--A customers  monthly transaction count is the count of all channels
--used to purchase items in the same day, over all days in the month.
--It is really a count of the minimum possible number of transactions
cust_purchase_cnt  AS
(  SELECT cust_id,  COUNT(*) cust_purchase_cnt
   FROM  cust_daily_trans_amt
   GROUP BY cust_id
),
--  Total profit for a customer over 1 month
cust_mon_profit AS
( SELECT cust_id, SUM(profit) cust_profit
       FROM  cust_prod_mon_profit
       GROUP BY cust_id
-- Minimum and maximum profit across all customer
-- sets endpoints for histogram data.
min_max_p AS
-- Note max profit + 0.1 to allow 10th bucket to include max value
(SELECT 0.1 + MAX(cust_profit) max_p, MIN(cust_profit) min_p
FROM cust_mon_profit),
-- Profitability bucket found for each customer
cust_bucket AS
(SELECT cust_id, cust_profit,
   width_bucket(cust_profit,
         min_max_p.min_p,
FROM cust_mon_profit,  min_max_p
-- Aggregated data needed for each bucket
histo_data AS
(  SELECT bucket,
     bucket*(( max_p-min_p) /10) top_end , count(*)  histo_count
   FROM  cust_bucket, min_max_p
   GROUP BY bucket, bucket*(( max_p - min_p) /10)
-- Median count of transactions per cust per month median_trans_count AS
-- Find median count of transactions per cust per month
(SELECT cust_bucket.bucket,
      PERCENTILE_CONT(0.5) WITHIN GROUP
          (ORDER BY cust_purchase_cnt.cust_purchase_cnt) median_trans_count
```

```
    FROM cust_bucket, cust_purchase_cnt
    WHERE cust_bucket.cust_id=cust_purchase_cnt.cust_id
    GROUP BY cust_bucket.bucket
-- Find Mmedian transaction size for custs by profit bucket
cust_median_trans_size AS
(  SELECT cust_bucket.bucket,
      PERCENTILE_CONT(0.5) WITHIN GROUP
          (ORDER BY cust_daily_trans_amt.cust_daily_trans_amt)
           cust_median_trans_ size
    FROM cust_bucket, cust_daily_trans_amt
    WHERE cust_bucket.cust_id=cust_daily_trans_amt.cust_id
    GROUP BY cust_bucket.bucket
-- Profitability of each product sold within each bucket
bucket_prod_profits AS
(  SELECT  cust_bucket.bucket, prod_id, SUM(profit) tot_prod_profit
   FROM  cust_bucket, cust_prod_mon_profit
   WHERE  cust_bucket.cust_id=cust_prod_mon_profit.cust_id
   GROUP BY cust_bucket.bucket, prod_id
),
-- Most and least profitable product by bucket
prod_profit AS
(  SELECT bucket, MIN(tot_prod_profit) min_profit_prod,
                  MAX(tot_prod_profit) max_profit_prod
   FROM bucket_prod_profits
   GROUP BY bucket
-- Main query block
SELECT  histo_data.bucket, histo_data.histo_count,
        median_trans_count.median_trans_count,
        cust_median_trans_size.cust_median_trans_size,
        prod_profit.min_profit_prod, prod_profit.max_profit_prod
FROM  histo_data, median_trans_count, cust_median_trans_size,
      prod_profit
WHERE  histo_data.bucket=median_trans_count.bucket
  AND  histo_data.bucket=cust_median_trans_size.bucket
  AND  histo_data.bucket=prod_profit.bucket;
```

Example 4 Frequent itemsets

Consider a marketing manager who wants to know which pieces of his firm's collateral are downloaded by users during a single session. That is, the manager wants to know which groupings of collateral are the most frequent itemsets. This is easy to do with the integrated frequent itemsets facility, as long as the Web site's activity log records a user ID and session ID for each collateral piece that is downloaded. For context, first we show a list of the aggregate number of downloads for individual white papers. (In our example data here, we use names of Oracle papers.)

```
White paper titles                                                #
------------------------------------------------------------      ----
Table Compression in Oracle Database 10g                           696
Field Experiences with Large Data Warehouses                       439
Key Data Warehouse Features: A Comparative Performance Analysis    181
Materialized Views in Oracle Database 10g                          167
Parallel Execution in Oracle Database 10g                          166
```

Here is a sample of the type of query that would be used for such analysis. The query uses `DBMS_FREQUENT_ITEMSET.FI_TRANSACTIONAL` as a table function. To understand the details of the query structure, see the *Oracle Database PL/SQL Packages and Types Reference*. The query returns the itemset of pairs of papers that were downloaded in a single session:

```
SELECT itemset, support, length, rnk
```

```
FROM
  (SELECT itemset, support, length,
   RANK(), OVER (PARTITION BY length ORDER BY support DESC) rnk
FROM
(SELECT CAST(itemset AS fi_char) itemset, support, length, total_tranx
 FROM table(DBMS_FREQUENT_ITEMSET.FI_TRANSACTIONAL
  (CURSOR(SELECT session_id, command
          FROM web_log
          WHERE time_stamp BETWEEN '01-APR-2002' AND '01-JUN-2002'),
          (60/2600), 2, 2, CURSOR(SELECT 'a' FROM DUAL WHERE 1=0),
          CURSOR(SELECT 'a' FROM DUAL WHERE 1=0)))))
   WHERE rnk <= 10;
```

Here are the first three items of results:

```
White paper titles                                                    #
-------------------------------------------------------------        -----
Table Compression in Oracle Database 10g                              115
Field Experiences with Large Data Warehouses

Data Warehouse Performance Enhancements with Oracle Database 10g      109
Oracle Performance and Scalability in DSS Environments

Materialized Views in Oracle Database 10g                             107
Query Optimization in Oracle Database 10g
```

This analysis yielded some interesting results. If one were to look at the list of the most popular single papers, one would expect the most popular pairs of downloaded papers would often include the white paper "Table Compression in Oracle Database 10*g*", because it was the most popular download of all papers. However, only one of the top three pairs included this paper.

By using frequent itemsets to analyze the Web log information, a manager can glean much more information than available in a simple report that only lists the most popular pieces of collateral. From these results, the manager can see that visitors to this Web site tend to search for information on a single topic area during a single session: visitors interested in scalability download white papers on compression and large data warehouses, while visitors interested in complex query capabilities download papers on query optimization and materialized views. For a marketing manager, this type of information is helpful in determining what sort of collateral should be written in the future; for a Web designer, this information can provide additional suggestions on how to organize the Web site.

See "Frequent Itemsets" on page 22-57 for more information.

Glossary

additive

Describes a **fact** (or **measure**) that can be summarized through addition. An additive fact is the most common type of fact. Examples include sales, cost, and profit. Contrast with **nonadditive** and **semi-additive**.

advisor

See **SQL Access Advisor**.

aggregate

Summarized data. For example, unit sales of a particular product could be aggregated by day, month, quarter and yearly sales.

aggregation

The process of consolidating data values into a single value. For example, sales data could be collected on a daily basis and then be aggregated to the week **level**, the week data could be aggregated to the month level, and so on. The data can then be referred to as **aggregate** data. The term aggregation is synonymous with summarization, and aggregate data is synonymous with summary data.

ancestor

A value at any **level** higher than a given value in a **hierarchy**. For example, in a Time **dimension**, the value 1999 might be the ancestor of the values Q1-99 and Jan-99.

attribute

A descriptive characteristic of one or more levels. For example, the product **dimension** for a clothing manufacturer might contain a **level** called item, one of whose attributes is color. Attributes represent logical groupings that enable end users to select data based on like characteristics.

Note that in relational modeling, an attribute is defined as a characteristic of an **entity**. In Oracle Database 10*g*, an attribute is a column in a **dimension** that characterizes each **element** of a single level.

cardinality

From an **OLTP** perspective, this refers to the number of rows in a table. From a data warehousing perspective, this typically refers to the number of distinct values in a column. For most **data warehouse** DBAs, a more important issue is the **degree of cardinality**.

change set

A set of logically grouped change data that is transactionally consistent. It contains one or more change tables.

change table

A relational table that contains change data for a single **source** table. To Change Data Capture **subscribers**, a change table is known as a **publication**.

child

A value at the **level** under a given value in a **hierarchy**. For example, in a Time **dimension**, the value Jan-99 might be the child of the value Q1-99. A value can be a child for more than one **parent** if the child value belongs to multiple hierarchies.

cleansing

The process of resolving inconsistencies and fixing the anomalies in **source** data, typically as part of the **ETL** process.

Common Warehouse Metadata (CWM)

A repository standard used by Oracle data warehousing, and decision support. The CWM repository **schema** is a standalone product that other products can share—each product owns only the objects within the CWM repository that it creates.

cross product

A procedure for combining the elements in multiple sets. For example, given two columns, each **element** of the first column is matched with every element of the second column. A simple example is illustrated as follows:

```
Col1   Col2   Cross Product
----   ----   -------------
a      c      ac
b      d      ad
              bc
              bd
```

Cross products are performed when grouping sets are concatenated, as described in Chapter 21, "SQL for Aggregation in Data Warehouses".

data mart

A **data warehouse** that is designed for a particular line of business, such as sales, marketing, or finance. In a dependent data mart, the data can be derived from an enterprise-wide data warehouse. In an independent data mart, data can be collected directly from sources.

data source

A database, application, repository, or file that contributes data to a warehouse.

data warehouse

A relational database that is designed for query and analysis rather than transaction processing. A data warehouse usually contains historical data that is derived from transaction data, but it can include data from other sources. It separates analysis workload from transaction workload and enables a business to consolidate data from several sources.

In addition to a relational database, a data warehouse environment often consists of an **ETL** solution, an analytical SQL engine, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

degree of cardinality

The number of unique values of a column divided by the total number of rows in the table. This is particularly important when deciding which indexes to build. You typically want to use bitmap indexes on low degree of cardinality columns and B-tree indexes on high degree of cardinality columns. As a general rule, a cardinality of under 1% makes a good candidate for a bitmap index.

denormalize

The process of allowing redundancy in a table. Contrast with **normalize**.

derived fact (or measure)

A **fact** (or **measure**) that is generated from existing data using a mathematical operation or a data **transformation**. Examples include averages, totals, percentages, and differences.

detail

See: **fact table**.

detail table

See: **fact table**.

dimension

The term dimension is commonly used in two ways:

- A general term for any characteristic that is used to specify the members of a data set. The three most common dimensions in a sales-oriented **data warehouse** are time, geography, and product. Most dimensions have hierarchies.
- An object defined in a database to enable queries to navigate dimensions. In Oracle Database 10*g*, a dimension is a database object that defines hierarchical (**parent**/**child**) relationships between pairs of column sets. In Oracle Express, a dimension is a database object that consists of a list of values.

dimension table

Dimension tables describe the business entities of an enterprise, represented as hierarchical, categorical information such as time, departments, locations, and products. Dimension tables are sometimes called lookup or reference tables.

dimension value

One **element** in the list that makes up a **dimension**. For example, a computer company might have dimension values in the product dimension called LAPPC and DESKPC. Values in the geography dimension might include Boston and Paris. Values in the time dimension might include MAY96 and JAN97.

drill

To navigate from one item to a set of related items. Drilling typically involves navigating up and down through a **level** (or levels) in a **hierarchy**. When selecting data, you expand a hierarchy when you **drill down** in it, and you collapse a hierarchy when you **drill up** in it.

drill down

To expand the view to include **child** values that are associated with **parent** values in the **hierarchy**.

drill up

To collapse the list of descendant values that are associated with a **parent** value in the **hierarchy**.

element

An object or process. For example, a **dimension** is an object, a **mapping** is a process, and both are elements.

entity

Entity is used in database modeling. In relational databases, it typically maps to a table.

ETL

ETL stands for **extraction**, **transformation**, and loading. ETL refers to the methods involved in accessing and manipulating **source** data and loading it into a **data warehouse**. The order in which these processes are performed varies.

Note that ETT (extraction, transformation, **transportation**) and ETM (extraction, transformation, move) are sometimes used instead of ETL.

extraction

The process of taking data out of a **source** as part of an initial phase of **ETL**.

fact

Data, usually numeric and **additive**, that can be examined and analyzed. Examples include sales, cost, and profit. Fact and measure are synonymous; fact is more commonly used with relational environments, measure is more commonly used with multidimensional environments. A **derived fact (or measure)** is generated from existing data using a mathematical operation or a data **transformation**.

fact table

A table in a **star schema** that contains facts. A fact table typically has two types of columns: those that contain facts and those that are **dimension table** foreign keys. The primary key of a fact table is usually a composite key that is made up of all of its foreign keys.

A fact table might contain either detail **level** facts or facts that have been aggregated (fact tables that contain aggregated facts are often instead called summary tables). A fact table usually contains facts with the same level of **aggregation**.

fast refresh

An operation that applies only the data changes to a **materialized view**, thus eliminating the need to rebuild the materialized view from scratch.

file-to-table mapping

Maps data from flat files to tables in the warehouse.

hierarchy

A logical structure that uses ordered levels as a means of organizing data. A hierarchy can be used to define data **aggregation**; for example, in a time **dimension**, a hierarchy

might be used to aggregate data from the Month **level** to the Quarter level to the Year level. Hierarchies can be defined in Oracle as part of the **dimension** object. A hierarchy can also be used to define a navigational **drill** path, regardless of whether the levels in the hierarchy represent aggregated totals.

high boundary

The newest row in a **subscription window**.

level

A position in a **hierarchy**. For example, a time **dimension** might have a hierarchy that represents data at the Month, Quarter, and Year levels.

level value table

A database table that stores the values or data for the levels you created as part of your dimensions and hierarchies.

low boundary

The oldest row in a **subscription window**.

mapping

The definition of the relationship and data flow between **source** and **target** objects.

materialized view

A pre-computed table comprising aggregated or joined data from **fact** and possibly a **dimension table**. Also known as a **summary** or **aggregate** table.

measure

See **fact**.

metadata

Data that describes data and other structures, such as objects, business rules, and processes. For example, the **schema** design of a **data warehouse** is typically stored in a repository as metadata, which is used to generate scripts used to build and populate the data warehouse. A repository contains metadata.

Examples include: for data, the definition of a **source** to **target transformation** that is used to generate and populate the data warehouse; for information, definitions of tables, columns and associations that are stored inside a relational modeling tool; for business rules, discount by 10 percent after selling 1,000 items.

model

An object that represents something to be made. A representative style, plan, or design. A model can also be **metadata** that defines the structure of the **data warehouse**.

nonadditive

Describes a **fact** (or **measure**) that cannot be summarized through addition. An example includes Average. Contrast with **additive** and **semi-additive**.

normalize

In a relational database, the process of removing redundancy in data by separating the data into multiple tables. Contrast with **denormalize**.

The process of removing redundancy in data by separating the data into multiple tables.

OLTP

See: **online transaction processing (OLTP)**.

online transaction processing (OLTP)

Online transaction processing. OLTP systems are optimized for fast and reliable transaction handling. Compared to **data warehouse** systems, most OLTP interactions will involve a relatively small number of rows, but a larger group of tables.

parallelism

Breaking down a task so that several processes do part of the work. When multiple CPUs each do their portion simultaneously, very large performance gains are possible.

parallel execution

Breaking down a task so that several processes do part of the work. When multiple CPUs each do their portion simultaneously, very large performance gains are possible.

parent

A value at the **level** above a given value in a **hierarchy**. For example, in a Time **dimension**, the value `Q1-99` might be the parent of the **child** value `Jan-99`.

partition

Very large tables and indexes can be difficult and time-consuming to work with. To improve manageability, you can break your tables and indexes into smaller pieces called partitions.

pivoting

A **transformation** where each record in an input stream is converted to many records in the appropriate table in the **data warehouse**. This is particularly important when taking data from nonrelational databases.

publication

A relational table that contains change data for a single **source** table. A Change Data Capture **publisher** refers to a publication as a **change table**.

publication ID

A publication ID is a unique numeric value that Change Data Capture assigns to each **change table** defined by a **publisher**.

publisher

Usually a database administrator who is in charge of creating and maintaining **schema** objects that make up the Change Data Capture system.

query rewrite

A mechanism to use a **materialized view** (which is precomputed) to quickly answer queries.

refresh

The mechanism whereby a **materialized view** is changed to reflect new data.

rewrite

See: **query rewrite**.

schema

A collection of related database objects. Relational schemas are grouped by database user ID and include tables, views, and other objects. The sample schemas `sh` are used throughout this Guide. Two special types of schema are **snowflake schema** and **star schema**.

semi-additive

Describes a **fact** (or **measure**) that can be summarized through addition along some, but not all, dimensions. Examples include headcount and on hand stock. Contrast with **additive** and **nonadditive**.

slice and dice

This is an informal term referring to data retrieval and manipulation. We can picture a **data warehouse** as a cube of data, where each axis of the cube represents a **dimension**. To "slice" the data is to retrieve a piece (a slice) of the cube by specifying measures and values for some or all of the dimensions. When we retrieve a data slice, we may also move and reorder its columns and rows as if we had diced the slice into many small pieces. A system with good slicing and dicing makes it easy to navigate through large amounts of data.

snowflake schema

A type of **star schema** in which each **dimension table** is partly or fully normalized.

source

A database, application, file, or other storage facility from which the data in a **data warehouse** is derived.

source system

A database, application, file, or other storage facility from which the data in a **data warehouse** is derived.

source tables

The tables in a **source** database.

SQL Access Advisor

The SQL Access Advisor helps you achieve your performance goals by recommending the proper **materialized view** set, materialized view logs, partitions, and indexes for a given workload. It is a GUI in Oracle Enterprise Manager, and has similar capabilities to the `DBMS_ADVISOR` package.

staging area

A place where data is processed before entering the warehouse.

staging file

A file used when data is processed before entering the warehouse.

star query

A join between a **fact table** and a number of dimension tables. Each **dimension table** is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other.

star schema

A relational **schema** whose design represents a multidimensional data **model**. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys.

subject area

A classification system that represents or distinguishes parts of an organization or areas of knowledge. A **data mart** is often developed to support a subject area such as sales, marketing, or geography.

subscribers

Consumers of the published change data. These are normally applications.

subscription

A mechanism for Change Data Capture **subscribers** that controls access to the change data from one or more **source tables** of interest within a single **change set**. A subscription contains one or more subscriber views.

subscription window

A mechanism that defines the range of rows in a Change Data Capture **publication** that the subscriber can currently see in subscriber views.

summary

See: **materialized view**.

Summary Advisor

Replaced by the **SQL Access Advisor**.

target

Holds the intermediate or final results of any part of the **ETL** process. The target of the entire ETL process is the **data warehouse**.

third normal form (3NF)

A classical relational database modeling technique that minimizes data redundancy through normalization.

third normal form schema

A **schema** that uses the same kind of normalization as typically found in an **OLTP** system. Third normal form schemas are sometimes chosen for a large **data warehouse**, especially an environment with significant data loading requirements that is used to feed a **data mart** and execute long-running queries. Compare with **snowflake schema** and **star schema**.

transformation

The process of manipulating data. Any manipulation beyond copying is a transformation. Examples include **cleansing**, aggregating, and integrating data from multiple **source tables**.

transportation

The process of moving copied or transformed data from a **source** to a **data warehouse**. Compare with **transformation**.

unique identifier

An identifier whose purpose is to differentiate between the same item when it appears in more than one place.

update window

The length of time available for updating a warehouse. For example, you might have 8 hours at night to update your warehouse.

update frequency

How often a **data warehouse** is updated with new information. For example, a warehouse might be updated nightly from an **OLTP** system.

validation

The process of verifying **metadata** definitions and configuration parameters.

versioning

The ability to create new versions of a **data warehouse** project for new requirements and changes.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X