Oracle® Database

Data Warehousing Guide

12*c* Release 1 (12.1)

E41670-08

November 2014

ORACLE®

Oracle Database Data Warehousing Guide, 12*c* Release 1 (12.1)

E41670-08

Primary Authors: Paul Lane, Padmaja Potineni

Contributors: Hermann Baer, Mark Bauer, Subhransu Basu, Nigel Bayliss, Donna Carver, Maria Colgan, Benoit Dageville, Luping Ding, Bud Endress, Bruce Golbus, John Haydu, Chun-Chieh Lin, William Lee, George Lumpkin, Alex Melidis, Valarie Moore, Ananth Raghavan, Jack Raitto, Lei Sheng, Wayne Smith, Sankar Subramanian, Margaret Taft, Murali Thiyagarajan, Jean-Francois Verrier, Andreas Walter, Andy Witkowski, Min Xiao, Tsae-Feng Yu, Fred Zemke, Mohamed Ziauddin

Contents

3 Data Warehousing Physical Design

4 Data Warehousing Optimizations and Techniques

Part II Optimizing Data Warehouses

5 Basic Materialized Views

6 Advanced Materialized Views

7 Refreshing Materialized Views

8 Synchronous Refresh

9 Dimensions

10 Basic Query Rewrite for Materialized Views

11 Advanced Query Rewrite for Materialized Views

12 Attribute Clustering

13 Using Zone Maps

17 Loading and Transformation in Data Warehouses

Part IV Relational Analytics

18 SQL for Analysis and Reporting

19 SQL for Aggregation in Data Warehouses

20 SQL for Pattern Matching

21 SQL for Modeling

22 Advanced Analytical SQL

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 that have purchased support 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.

Changes in This Release for Oracle Database Data Warehousing Guide

This chapter contains:

Changes in Oracle Database 12*c* Release 1 (12.1.0.2)

The following are the changes in *Oracle Database Data Warehousing Guide* for Oracle Database 12*c* Release 1 (12.1.0.2):

New Features

- Oracle In-Memory Column Store

  The Oracle In-Memory Column Store (IM column store) in an optional area in the SGA that stores tables, table partitions, and individual columns in a compressed columnar format. The IM column store is a supplement to rather than a replacement for the database buffer cache.

  The IM column store primarily improves the performance of table scans and the application of `WHERE` clause predicates. Faster table scans make it more likely that the optimizer will choose bloom filters and `VECTOR GROUP BY` transformations.

  See Also:

  - "About the Oracle In-Memory Column Store" on page 2-11
  - "In-Memory Aggregation" on page 19-22

- Attribute clustering

  Attribute clustering of tables enables you to store data in close proximity on disk in a ordered way that is based on the values of certain columns in the table. I/O and CPU costs of table scans and table data lookup through indexes are reduced because pruning though table zone maps becomes more effective.

  See Also: Chapter 12, "Attribute Clustering"

- Zone maps

  Zone maps enable natural pruning of data based on physical location of the data on disk. Accessing only the relevant data blocks during full table scans and

accessing only the relevant data rows during index scans reduces I/O and CPU costs of data access.

See Also: Chapter 13, "Using Zone Maps"

- In-memory aggregation

  The `VECTOR GROUP BY` operation improves the performance of queries that join one or more relatively small tables to a larger table and aggregate data. In the context of data warehousing, `VECTOR GROUP BY` aggregation will often be chosen for star queries that select data from the IM column store.

  `VECTOR GROUP BY` aggregation minimizes the processing involved in joining multiple dimension tables to one fact table. It uses the infrastructure related to parallel query and blends it with CPU-efficient algorithms that maximize performance.

  See Also: "About In-Memory Aggregation" on page 2-16

- Automatic Big Table Caching

  Automatic big table caching improves in-memory query performance for large tables that do not fit completely in the buffer cache. Such tables can be stored in the big table cache, an optional, configurable portion of the database buffer cache.

  See Also: "Automatic Big Table Caching to Improve the Performance of In-Memory Parallel Queries" on page 2-15

Changes in Oracle Database 12*c* Release 1 (12.1.0.1)

The following are changes in *Oracle Database Data Warehousing Guide* for Oracle Database 12*c* Release 1 (12.1.0.1).

New Features

- Pattern Matching

  SQL has been extended to support pattern matching, which makes it easy to detect various patterns over sequences. Pattern matching is useful in many commercial applications, such as stock monitoring, network intrusion detection, and e-commerce purchase tracking.

  See Also: Chapter 20, "SQL for Pattern Matching" for more information

- Native SQL Support for Top-N Queries

  The new *`row_limiting_clause`* enables you to limit the rows returned by a query. You can specify an offset, and number of rows or percentage of rows to return. This enables you to implement top-N reporting.

  See Also: "Limiting SQL Rows" on page 18-60 for more information

- Online Statistics Gathering for Bulk Load Operations

  Starting in Oracle Database 12*c*, the database automatically gathers table statistics as part of bulk load operations.

- Synchronous Refresh

A new type of refresh called synchronous refresh enables you to keep a set of tables and materialized views defined on them to always be in sync. It is well suited for data warehouses where the loading of incremental data is tightly controlled and occurs at periodic intervals.

> **See Also:** Chapter 8, "Synchronous Refresh" for more information

- Out-of-Place Refresh

  A new type of refresh is available to improve materialized view refresh performance and availability. This refresh, called out-of-place refresh because it uses outside tables during refresh, is particularly effective when handling situations with large amounts of data changes, where conventional DML statements do not scale well.

  > **See Also:** Chapter 7, "Refreshing Materialized Views" for more information

Desupported Features

Some features previously described in this document are desupported in Oracle Database 12*c* Release 1. See *Oracle Database Upgrade Guide* for a list of desupported features.

Part I

Data Warehouse - Fundamentals

This section introduces basic data warehousing concepts.

It contains the following chapters:

1

Introduction to Data Warehousing Concepts

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

What Is a Data Warehouse?

A **data warehouse** is a database designed to enable business intelligence activities: it exists to help users understand and enhance their organization's performance. It is designed for query and analysis rather than for transaction processing, and 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.

To achieve the goal of enhanced business intelligence, the data warehouse works with data collected from multiple sources. The source data may come from internally developed systems, purchased applications, third-party data syndicators and other sources. It may involve transactions, production, marketing, human resources and more. In today's world of big data, the data may be many billions of individual clicks on web sites or the massive data streams from sensors built into complex machinery.

Data warehouses are distinct from online transaction processing (OLTP) systems. With a data warehouse you separate analysis workload from transaction workload. Thus data warehouses are very much read-oriented systems. They have a far higher amount of data reading versus writing and updating. This enables far better analytical performance and avoids impacting your transaction systems. A data warehouse system can be optimized to consolidate data from many sources to achieve a key goal: it becomes your organization's "single source of truth". There is great value in having a

consistent source of data that all users can look to; it prevents many disputes and enhances decision-making efficiency.

A data warehouse usually stores many months or years of data to support historical analysis. The data in a data warehouse is typically loaded through an extraction, transformation, and loading (ETL) process from multiple data sources. Modern data warehouses are moving toward an extract, load, transformation (ELT) architecture in which all or most data transformation is performed on the database that hosts the data warehouse. It is important to note that defining the ETL process is a very large part of the design effort of a data warehouse. Similarly, the speed and reliability of ETL operations are the foundation of the data warehouse once it is up and running.

Users of the data warehouse perform data analyses that are often time-related. Examples include consolidation of last year's sales figures, inventory analysis, and profit by product and by customer. But time-focused or not, users want to "slice and dice" their data however they see fit and a well-designed data warehouse will be flexible enough to meet those demands. Users will sometimes need highly aggregated data, and other times they will need to drill down to details. More sophisticated analyses include trend analyses and data mining, which use existing data to forecast trends or predict futures. The data warehouse acts as the underlying engine used by middleware business intelligence environments that serve reports, dashboards and other interfaces to end users.

Although the discussion above has focused on the term "data warehouse", there are two other important terms that need to be mentioned. These are the data mart and the operation data store (ODS).

A data mart serves the same role as a data warehouse, but it is intentionally limited in scope. It may serve one particular department or line of business. The advantage of a data mart versus a data warehouse is that it can be created much faster due to its limited coverage. However, data marts also create problems with inconsistency. It takes tight discipline to keep data and calculation definitions consistent across data marts. This problem has been widely recognized, so data marts exist in two styles. Independent data marts are those which are fed directly from source data. They can turn into islands of inconsistent information. Dependent data marts are fed from an existing data warehouse. Dependent data marts can avoid the problems of inconsistency, but they require that an enterprise-level data warehouse already exist.

Operational data stores exist to support daily operations. The ODS data is cleaned and validated, but it is not historically deep: it may be just the data for the current day. Rather than support the historically rich queries that a data warehouse can handle, the ODS gives data warehouses a place to get access to the most current data, which has not yet been loaded into the data warehouse. The ODS may also be used as a source to load the data warehouse. As data warehousing loading techniques have become more advanced, data warehouses may have less need for ODS as a source for loading data. Instead, constant trickle-feed systems can load the data warehouse in near real time.

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 Varient

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 Varient

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.

Key Characteristics of a Data Warehouse

The key characteristics of a data warehouse are as follows:

- Data is structured for simplicity of access and high-speed query performance.
- End users are time-sensitive and desire speed-of-thought response times.
- Large amounts of historical data are used.
- Queries often retrieve large amounts of data, perhaps many thousands of rows.
- Both predefined and ad hoc queries are common.
- The data load involves multiple sources and transformations.

In general, fast query performance with high data throughput is the key to a successful data warehouse.

Contrasting OLTP and Data Warehousing Environments

There are important differences between an OLTP system and a data warehouse. One major difference between the types of system is that data warehouses are not exclusively 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 partially denormalized schemas 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.

Common Data Warehouse Tasks

As an Oracle data warehousing administrator or designer, you can expect to be involved in the following tasks:

- Configuring an Oracle database for use as a data warehouse
- Designing data warehouses
- Performing upgrades of the database and data warehousing software to new releases
- Managing schema objects, such as tables, indexes, and materialized views
- Managing users and security
- Developing routines used for the extraction, transformation, and loading (ETL) processes
- Creating reports based on the data in the data warehouse

- Backing up the data warehouse and performing recovery when necessary
- Monitoring the data warehouse's performance and taking preventive or corrective action as required

In a small-to-midsize data warehouse environment, you might be the sole person performing these tasks. In large, enterprise environments, the job is often divided among several DBAs and designers, each with their own specialty, such as database security or database tuning.

These tasks are illustrated in the following:

- For more information regarding partitioning, see *Oracle Database VLDB and Partitioning Guide*.
- For more information regarding database security, see *Oracle Database Security Guide*.
- For more information regarding database performance, see *Oracle Database Performance Tuning Guide* and *Oracle Database SQL Tuning Guide*.
- For more information regarding backup and recovery, see *Oracle Database Backup and Recovery User's Guide*.
- For more information regarding ODI, see *Oracle Fusion Middleware Developer's Guide for Oracle Data Integrator*.

Data Warehouse Architectures

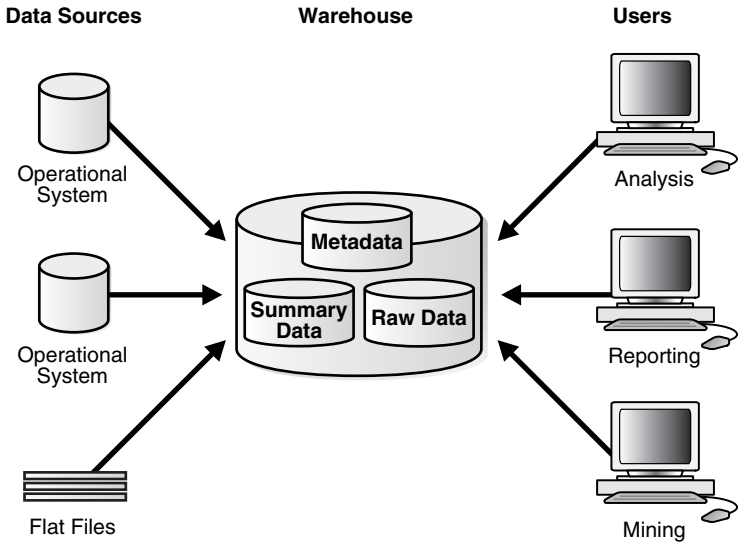
Data warehouses and their architectures vary depending upon the specifics of an organization's situation. Three common architectures are:

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

Figure 1–1 Architecture of a Data Warehouse



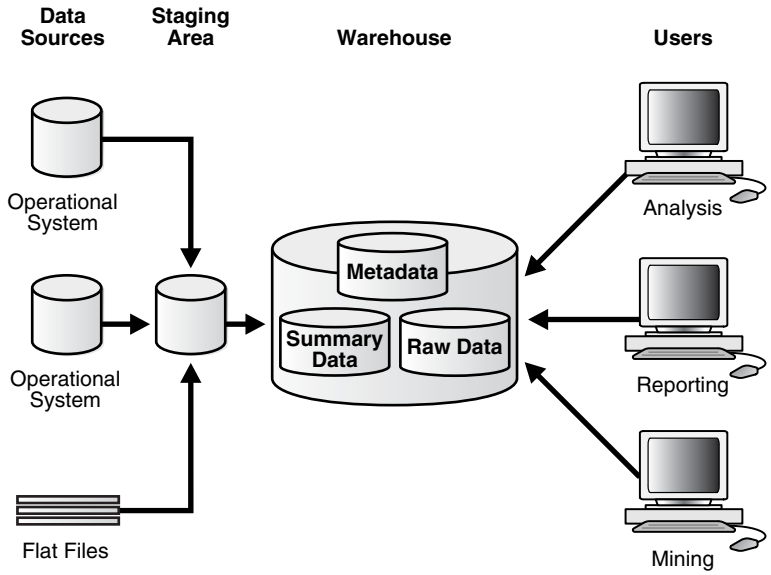
In Figure 1–1, the metadata and raw data of a traditional OLTP system is present, as is an additional type of data, summary data. Summaries are a mechanism to pre-compute common expensive, long-running operations for sub-second data retrieval. For example, a typical data warehouse query is to retrieve something such as August sales. A summary in an Oracle database is called a **materialized view**.

The consolidated storage of the raw data as the center of your data warehousing architecture is often referred to as an Enterprise Data Warehouse (EDW). An EDW provides a 360-degree view into the business of an organization by holding all relevant business information in the most detailed format.

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 1–2. You can do this programmatically, although most data warehouses use a **staging area** instead. A staging area simplifies data cleansing and consolidation for operational data coming from multiple source systems, especially for enterprise data warehouses where all relevant information of an enterprise is consolidated. Figure 1–2 illustrates this typical architecture.

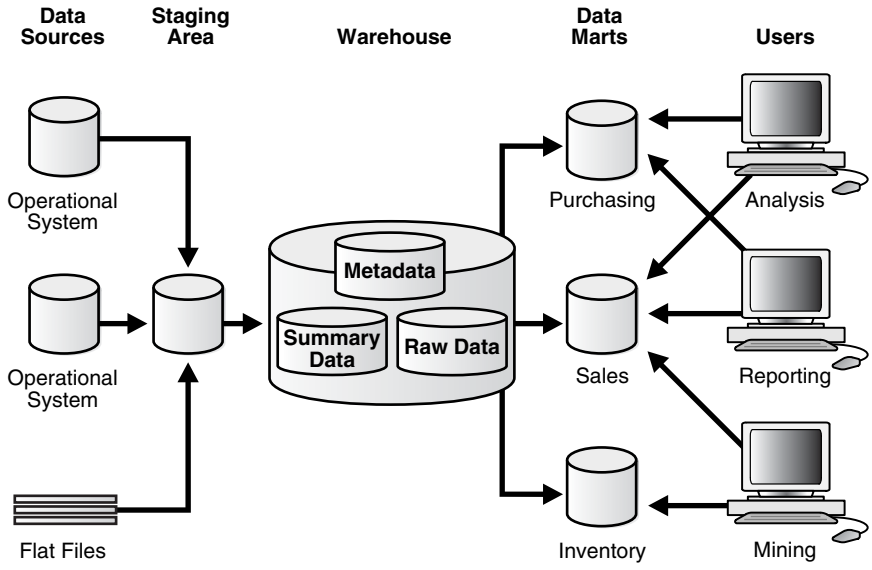
Figure 1–2 Architecture of a Data Warehouse with a Staging Area



Data Warehouse Architecture: with a Staging Area and Data Marts

Although the architecture in Figure 1–2 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 1–3 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 1–3 Architecture of a Data Warehouse with a Staging Area and Data Marts



Note: Data marts can be physically instantiated or implemented purely logically though views. Furthermore, data marts can be co-located with the enterprise data warehouse or built as separate systems. Building an end-to-end data warehousing architecture with an enterprise data warehouse and surrounding data marts is not the focus of this book.

2

Data Warehousing Logical Design

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 an enterprise data warehouse. You have defined the business requirements and agreed upon the scope of your business goals, 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.

Entity-relationship modeling is purely logical and applies to both OLTP and data warehousing systems. It is also applicable to the various common physical schema modeling techniques found in data warehousing environments, namely normalized (3NF) schemas in Enterprise Data Warehousing environments, star or snowflake schemas in data marts, or hybrid schemas with components of both of these classical modeling techniques.

See Also:

- *Oracle Fusion Middleware Developer's Guide for Oracle Data Integrator* for more details regarding ODI

What is a Schema?

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. A key part of designing the schema is whether to use a third normal form, star, or snowflake schema, and these are discussed later.

About Third Normal Form Schemas

Third Normal Form design seeks to minimize data redundancy and avoid anomalies in data insertion, updates and deletion. 3NF design has a long heritage in online transaction processing (OLTP) systems. OLTP systems must maximize performance and accuracy when inserting, updating and deleting data. Transactions must be handled as quickly as possible or the business may be unable to handle the flow of events, perhaps losing sales or incurring other costs. Therefore, 3NF designs avoid redundant data manipulation and minimize table locks, both of which can slow

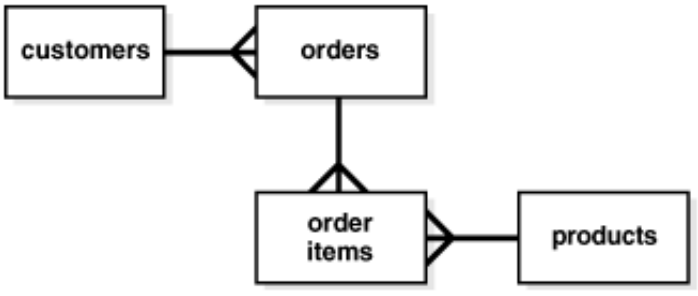
inserts, updates and deletes. 3NF designs also works well to abstract the data from specific application needs. If new types of data are added to the environment, you can extend the data model with relative ease and minimal impact to existing applications. Likewise, if you have completely new types of analyses to perform in your data warehouse, a well-designed 3NF schema will be able to handle them without requiring redesigned data structures.

3NF designs have great flexibility, but it comes at a cost. 3NF databases use very many tables and this requires complex queries with many joins. For full scale enterprise models built in 3NF form, over one thousand tables are commonly encountered in the schema. With the kinds of queries involved in data warehousing, which will often need access to many rows from many tables, this design imposes understanding and performance penalties. It can be complex for query builders, whether they are humans or business intelligence tools and applications, to choose and join the tables needed for a given piece of data when there are very large numbers of tables available. Even when the tables are readily chosen by the query generator, the 3NF schema often requires that a large number of tables be used in a single query. More tables in a query mean more potential data access paths, which makes the database query optimizer's job harder. The end result can be slow query performance.

The issue of slow query performance in a 3NF system is not necessarily limited to the core queries used to create reports and analyses. It can also show up in the simpler task of users browsing subsets of data to understand the contents. Similarly, the complexity of a 3NF schema may impact generating the pick-lists of data used to constrain queries and reports. Although these may seem relatively minor issues, speedy response time for such processes makes a big impact on user satisfaction.

Figure 2–1 presents a tiny fragment of a 3NF Schema. Note how order information is broken into order and order items to avoid redundant data storage. The "crow's feet" markings on the relationship between tables indicate one-to-many relationships among the entities. Thus, one order may have multiple order items, a single customer may have many orders, and a single product may be found in many order items. Although this diagram shows a very small case, you can see that minimizing data redundancy can lead to many tables in the schema.

Figure 2–1 Fragment of a Third Normal Form Schema



This section contains the following topics:

- About Normalization
- Design Concepts for 3NF Schemas

About Normalization

Normalization is a data design process that has a high level goal of keeping each fact in just one place to avoid data redundancy and insert, update, and delete anomalies. There are multiple levels of normalization, and this section describes the first three of

them. Considering how fundamental the term third normal form (3NF) term is, it only makes sense to see how 3NF is reached.

Consider a situation where you are tracking sales. The core entity you track is sales orders, where each sales order contains details about each item purchased (referred to as a line item): its name, price, quantity, and so on. The order also holds the name and address of the customer and more. Some orders have many different line items, and some orders have just one.

In first normal form (1NF), there are no repeating groups of data and no duplicate rows. Every intersection of a row and column (a field) contains just one value, and there are no groups of columns that contain the same facts. To avoid duplicate rows, there is a primary key. For sales orders, in first normal form, multiple line items of each sales order in a single field of the table are not displayed. Also, there will not be multiple columns showing line items.

Then comes second normal form (2NF), where the design is in first normal form and every non-key column is dependent on the complete primary key. Thus, the line items are broken out into a table of sales order line items where each row represents one line item of one order. You can look at the line item table and see that the names of the items sold are not dependent on the primary key of the line items table: the sales item is its own entity. Therefore, you move the sales item to its own table showing the item name. Prices charged for each item can vary by order (for instance, due to discounts) so these remain in the line items table. In the case of sales order, the name and address of the customer is not dependent on the primary key of the sales order: customer is its own entity. Thus, you move the customer name and address columns out into their own table of customer information.

Next is third normal form, where the goal is to ensure that there are no dependencies on non-key attributes. So the goal is to take columns that do not directly relate to the subject of the row (the primary key), and put them in their own table. So details about customers, such as customer name or customer city, should be put in a separate table, and then a customer foreign key added into the orders table.

Another example of how a 2NF table differs from a 3NF table would be a table of the winners of tennis tournaments that contained columns of tournament, year, winner, and winner's date of birth. In this case, the winner's date of birth is vulnerable to inconsistencies, as the same person could be shown with different dates of birth in different records. The way to avoid this potential problem is to break the table into one for tournament winners, and another for the player dates of birth.

Design Concepts for 3NF Schemas

The following section discusses some basic concepts when modeling for a data warehousing environment using a 3NF schema approach. The intent is not to discuss the theoretical foundation for 3NF modeling (or even higher levels of normalization), but to highlight some key components relevant for data warehousing.

This section contains the following topics:

- Identifying Candidate Primary Keys
- Foreign Key Relationships and Referential Integrity Constraints
- Denormalization

Identifying Candidate Primary Keys

A primary key is an attribute that uniquely identifies a specific record in a table. Primary keys can be identified through single or multiple columns. It is normally

preferred to achieve unique identification through as little columns as possible - ideally one or two - and to either use a column that is most likely not going to be updated or even changed in bulk. If your data model does not lead to a simple unique identification through its attributes, you would require too many attributes to uniquely identify a single records, or the data is prone to changes, the usage of a surrogate key is highly recommended.

Specifically, 3NF schemas rely on proper and simple unique identification since queries tend to have many table joins and all columns necessary to uniquely identify a record are needed as join condition to avoid row duplication through the join.

Foreign Key Relationships and Referential Integrity Constraints

3NF schemas in data warehousing environments often resemble the data model of its OLTP source systems, in which the logical consistency between data entities is expressed and enforced through primary key - foreign key relationships, also known as parent-child relationship. A foreign key resolves a 1-to-many relationship in relational system and ensures logical consistency: for example, you cannot have an order line item without an order header, or an employee working for a non-existent department.

While such referential are always enforced in OLTP system, data warehousing systems often implement them as declarative, non-enforced conditions, relying on the ETL process to ensure data consistency. Whenever possible, foreign keys and referential integrity constraints should be defined as non-enforced conditions, since it enables better query optimization and cardinality estimates.

Denormalization

Proper normalized modelling tends to decompose logical entities - such as a customer. a product, or an order - into many physical tables, making even the retrieval of perceived simple information requiring to join many tables. While this is not a problem from a query processing perspective, it can put some unnecessary burden on both the application developer (for writing code) as well as the database (for joining information that is always used together). It is not uncommon to see some sensible level of denormalization in 3NF data warehousing models, in a logical form as views or in a physical form through slightly denormalized tables.

Care has to be taken with the physical denormalization to preserve the subject-neutral shape and therefore the flexibility of the physical implementation of the 3NF schema.

About Star Schemas

Star schemas are often found in data warehousing systems with embedded logical or physical data marts. The term star schema is another way of referring to a "dimensional modeling" approach to defining your data model. Most descriptions of dimensional modeling use terminology drawn from the work of Ralph Kimball, the pioneering consultant and writer in this field. Dimensional modeling creates multiple star schemas, each based on a business process such as sales tracking or shipments. Each star schema can be considered a data mart, and perhaps as few as 20 data marts can cover the business intelligence needs of an enterprise. Compared to 3NF designs, the number of tables involved in dimensional modeling is a tiny fraction. Many star schemas will have under a dozen tables. The star schemas are knit together through conformed dimensions and conformed facts. Thus, users are able to get data from multiple star schemas with minimal effort.

The goal for star schemas is structural simplicity and high performance data retrieval. Because most queries in the modern era are generated by reporting tools and

applications, it's vital to make the query generation convenient and reliable for the tools and application. In fact, many business intelligence tools and applications are designed with the expectation that a star schema representation will be available to them.

Discussions of star schemas are less abstracted from the physical database than 3NF descriptions. This is due to the pragmatic emphasis of dimensional modeling on the needs of business intelligence users.

Note how different the dimensional modeling style is from the 3NF approach that minimizes data redundancy and the risks of update/inset/delete anomalies. The star schema accepts data redundancy (denormalization) in its dimension tables for the sake of easy user understanding and better data retrieval performance. A common criticism of star schemas is that they limit analysis flexibility compared to 3NF designs. However, a well designed dimensional model can be extended to enable new types of analysis, and star schemas have been successful for many years at the largest enterprises.

As noted earlier, the modern approach to data warehousing does not pit star schemas and 3NF against each other. Rather, both techniques are used, with a foundation layer of 3NF - the Enterprise Data Warehouse of 3NF, acting as the bedrock data, and star schemas as a central part of an access and performance optimization layer.

This section contains the following topics:

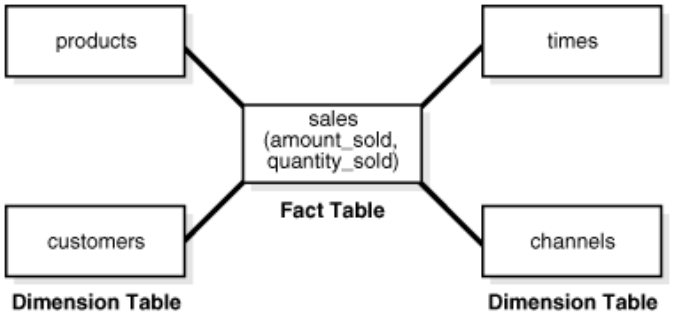
- Facts and Dimensions
- Design Concepts in Star Schemas
- About Snowflake Schemas

Facts and Dimensions

Star schemas divide data into facts and dimensions. Facts are the measurements of some event such as a sale and are typically numbers. Dimensions are the categories you use to identify facts, such as date, location, and product.

The name "star schema" comes from the fact that the diagrams of the schemas typically show a central fact table with lines joining it to the dimension tables, so the graphic impression is similar to a star. Figure 2–2 is a simple example with sales as the fact table and products, times, customers, and channels as the dimension table.

Figure 2–2 Star Schema



This section contains the following topics:

- Fact Tables
- Dimension Tables

Fact Tables

Fact tables have measurement data. They have many rows but typically not many columns. Fact tables for a large enterprise can easily hold billions of rows. For many star schemas, the fact table will represent well over 90 percent of the total storage space. A fact table has a composite key made up of the primary keys of the dimension tables of the schema.

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 stored in physical warehouses, where you may be able to add across a dimension of warehouse sites, but you cannot aggregate across time.

In terms of adding rows to data in a fact table, there are three main approaches:

- Transaction-based

  Shows a row for the finest level detail in a transaction. A row is entered only if a transaction has occurred for a given combination of dimension values. This is the most common type of fact table.

- Periodic Snapshot

  Shows data as of the end of a regular time interval, such as daily or weekly. If a row for the snapshot exists in a prior period, a row is entered for it in the new period even if no activity related to it has occurred in the latest interval. This type of fact table is useful in complex business processes where it is difficult to compute snapshot values from individual transaction rows.

- Accumulating Snapshot

  Shows one row for each occurrence of a short-lived process. The rows contain multiple dates tracking major milestones of a short-lived process. Unlike the other two types of fact tables, rows in an accumulating snapshot are updated multiple times as the tracked process moves forward.

Dimension Tables

Dimension tables provide category data to give context to the fact data. For instance, a star schema for sales data will have dimension tables for product, date, sales location, promotion and more. Dimension tables act as lookup or reference tables because their information lets you choose the values used to constrain your queries. The values in many dimension tables may change infrequently. As an example, a dimension of geographies showing cities may be fairly static. But when dimension values do change, it is vital to update them fast and reliably. Of course, there are situations where data warehouse dimension values change frequently. The customer dimension for an enterprise will certainly be subject to a frequent stream of updates and deletions.

A key aspect of dimension tables is the hierarchy information they provide. Dimension data typically has rows for the lowest level of detail plus rows for aggregated dimension values. These natural rollups or aggregations within a dimension table are called hierarchies and add great value for analyses. For instance, if you want to calculate the share of sales that a specific product represents within its specific product category, it is far easier and more reliable to have a predefined hierarchy for product aggregation than to specify all the elements of the product category in each query.

Because hierarchy information is so valuable, it is common to find multiple hierarchies reflected in a dimension table.

Dimension tables are usually textual and descriptive, and you will use their values as the row headers, column headers and page headers of the reports generated by your queries. While dimension tables have far fewer rows than fact tables, they can be quite wide, with dozens of columns. A location dimension table might have columns indicating every level of its rollup hierarchy, and may show multiple hierarchies reflected in the table. The location dimension table could have columns for its geographic rollup, such as street address, postal code, city, state/province, and country. The same table could include a rollup hierarchy set up for the sales organization, with columns for sales district, sales territory, sales region, and characteristics.

> **See Also:** Chapter 9, "Dimensions" for further information regarding dimensions

Design Concepts in Star Schemas

Here we touch on some of the key terms used in star schemas. This is by no means a full set, but is intended to highlight some of the areas worth your consideration.

This section contains the following topics:

- Data Grain
- Working with Multiple Star Schemas
- Conformed Dimensions
- Conformed Facts
- Surrogate Keys
- Degenerate Dimensions
- Junk Dimensions
- Embedded Hierarchy
- Factless Fact Tables
- Slowly Changing Dimensions

Data Grain

One of the most important tasks when designing your model is to consider the level of detail it will provide, referred to as the grain of the data. Consider a sales schema: will the grain be very fine, storing every single item purchased by each customer? Or will it be a coarse grain, storing only the daily totals of sales for each product at each store? In modern data warehousing there is a strong emphasis on providing the finest grain data possible, because this allows for maximum analytic power. Dimensional modeling experts generally recommend that each fact table store just one grain level. Presenting fact data in single-grain tables supports more reliable querying and table maintenance, because there is no ambiguity about the scope of any row in a fact table.

Working with Multiple Star Schemas

Because the star schema design approach is intended to chunk data into distinct processes, you need reliable and performant ways to traverse the schemas when queries span multiple schemas. One term for this ability is a data warehouse bus

architecture. A data warehouse bus architecture can be achieved with conformed dimensions and conformed facts.

Conformed Dimensions

Conformed dimensions means that dimensions are designed identically across the various star schemas. Conformed dimensions use the same values, column names and data types consistently across multiple stars. The conformed dimensions do not have to contain the same number of rows in each schema's copy of the dimension table, as long as the rows in the shorter tables are a true subset of the larger tables.

Conformed Facts

If the fact columns in multiple fact tables have exactly the same meaning, then they are considered conformed facts. Such facts can be used together reliably in calculations even though they are from different tables. Conformed facts should have the same column names to indicate their conformed status. Facts that are not conformed should always have different names to highlight their different meanings.

Surrogate Keys

Surrogate or artificial keys, usually sequential integers, are recommended for dimension tables. By using surrogate keys, the data is insulated from operational changes. Also, compact integer keys may allow for better performance than large and complex alphanumeric keys.

Degenerate Dimensions

Degenerate dimensions are dimension columns in fact tables that do not join to a dimension table. They are typically items such as order numbers and invoice numbers. You will see them when the grain of a fact table is at the level of an order line-item or a single transaction.

Junk Dimensions

Junk dimensions are abstract dimension tables used to hold text lookup values for flags and codes in fact tables. These dimensions are referred to as junk, not because they have low value, but because they hold an assortment of columns for convenience, analogous to the idea of a "junk drawer" in your home. The number of distinct values (cardinality) of each column in a junk dimension table is typically small.

Embedded Hierarchy

Classic dimensional modeling with star schemas advocates that each table contain data at a single grain. However, there are situations where designers choose to have multiple grains in a table, and these commonly represent a rollup hierarchy. A single sales fact table, for instance, might contain both transaction-level data, then a day-level rollup by product, then a month-level rollup by product. In such cases, the fact table will need to contain a level column indicating the hierarchy level applying to each row, and queries against the table will need to include a level predicate.

Factless Fact Tables

Factless fact tables do not contain measures such as sales price or quantity sold. Instead, the rows of a factless fact table are used to show events not represented by other fact tables. Another use for factless tables is as a "coverage table" which holds all the possible events that could have occurred in a given situation, such as all the

products that were part of a sales promotion and might have been sold at the promotional price.

Slowly Changing Dimensions

One of the certainties of data warehousing is that the way data is categorized will change. Product names and category names will change. Characteristics of a store will change. The areas included in sales territories will change. The timing and extent of these changes will not always be predictable. How can these slowly changing dimensions be handled? Star schemas treat these in three main ways:

- Type 1

  The dimension values that change are simply overwritten, with no history kept. This creates a problem for time-based analyses. Also, it invalidates any existing aggregates that depended on the old value of the dimension.

- Type 2

  When a dimension value changes, a new dimension row showing the new value and having a new surrogate key is created. You may choose to include date columns in our dimension showing when the new row is valid and when it is expired. No changes need be made to the fact table.

- Type 3

  When a dimension value is changed, the prior value is stored in a different column of the same row. This enables easy query generation if you want to compare results using the current and prior value of the column.

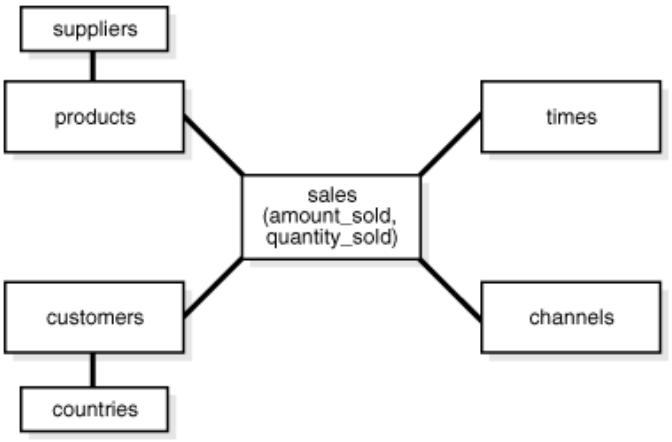
In practice, Type 2 is the most common treatment for slowly changing dimensions.

About 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 2–3 presents a graphical representation of a snowflake schema.

Figure 2–3 Snowflake Schema



About the Oracle In-Memory Column Store

Note: This feature is available starting with Oracle Database 12*c* Release 1 (12.1.0.2).

Traditional analytics has certain limitations or requirements that need to be managed to obtain good performance for analytic queries. You need to know user access patterns and then customize your data structures to provide optimal performance for these access patterns. Existing indexes, materialized views, and OLAP cubes need to be tuned. Certain data marts and reporting databases have complex ETL and thus need specialized tuning. Additionally, you need to strike a balance between performing analytics on stale data and slowing down OLTP operations on the production databases.

The Oracle In-Memory Column Store (IM column store) within the Oracle Database provides improved performance for both ad-hoc queries and analytics on live data. The live transactional database is used to provide instant answers to queries, thus enabling you to seamlessly use the same database for OLTP transactions and data warehouse analytics.

The IM column store is an optional area in the SGA that stores copies of tables, table partitions, and individual columns in a compressed columnar format that is optimized for rapid scans. Columnar format lends itself to easily to vector processing thus making aggregations, joins, and certain types of data retrieval faster than the traditional on-disk formats. The columnar format exists only in memory and does not replace the on-disk or buffer cache format. Instead, it supplements the buffer cache and provides an additional, transaction-consistent, copy of the table that is independent of the disk format.

See Also:

- Benefits of Using the Oracle In-Memory Column Store
- *Oracle Database Concepts* for conceptual information the about IM column store

Configuring the Oracle In-Memory Column Store

Configuring the IM column store is simple. You set the `INMEMORY_SIZE` initialization parameter, and then use DDL to specify the tablespaces, tables, partitions, or columns to be populated into the IM column store.

> **See Also:** *Oracle Database Administrator's Guide* for information about configuring the IM column store

Populating the Oracle In-Memory Column Store

You can specify that the database populates data into the IM column store from row storage either at database instance startup or when the data is accessed for the first time.

> **See Also:** *Oracle Database Concepts* for detailed information about how the IM column store is populated

In-Memory Columnar Compression

The Oracle Databases uses special compression formats that are optimized for access speeds rather than storage reductions to store data in the IM column store. You can select different compression options for each table, partition, or column.

> **See Also:**
>
> - *Oracle Database Concepts*
> - *Oracle Database Administrator's Guide*

Benefits of Using the Oracle In-Memory Column Store

The IM column store enables the Oracle Database to perform scans, joins, and aggregates much faster than when it uses the on-disk format exclusively. Business applications, ad-hoc analytic queries, and data warehouse workloads benefit most. Pure OLTP databases that perform short transactions using index lookups benefit less.

The IM column store seamlessly integrates with the Oracle Database. All existing database features, including High Availability features, are supported with no application changes required. Therefore, by configuring the IM column store, you can instantly improve the performance of existing analytic workloads and ad-hoc queries.

The Oracle Optimizer is aware of the IM column store making it possible for the Oracle Database to seamlessly send analytic queries to the IM column store while OLTP queries and DML are sent to the row store.

The advantages offered by the IM column store for data warehousing environments are:

- Faster scanning of large number of rows and applying filters that use operators such as =,<,>, and IN.
- Faster querying of a subset of columns in a table, for example, selecting 5 of 100 columns. See "Faster Performance for Analytic Queries" on page 2-13.
- Enhanced performance for joins by converting predicates on small dimension tables to filters on a large fact table. See "Enhanced Join Performance Using Vector Joins" on page 2-13.
- Efficient aggregation by using `VECTOR GROUP BY` transformation and vector array processing. See "Enhanced Aggregation Using VECTOR GROUP BY Transformations" on page 2-13.

- Reduced storage space and significantly less processing overhead because fewer indexes, materialized views, and OLAP cubes are required when IM column store is used.

  See Also: *Oracle Database Concepts* for information about the other advantages of using IM column store

Faster Performance for Analytic Queries

Storing data in memory using columnar format provides fast throughput for analyzing large amounts of data. This is useful for ad-hoc queries with unanticipated access patterns. Columnar format uses fixed-width columns for most numeric and short string data types. This enables very fast vector processing that answers queries faster. Only the columns necessary for the specific data analysis task are scanned instead of entire rows of data.

Data can be analyzed in real-time, thus enabling you to explore different possibilities and perform iteration. Using the IM column store requires fewer OLAP cubes to be created to obtain query results.

For example, you need to find the number of sales in the state of California this year. This data is stored in the `SALES` table. When this table is stored in the IM column store, the database needs to just scan the State column and count the number of occurrences of state California.

Enhanced Join Performance Using Vector Joins

IM column store takes advantage of vector joins. Vector joins speed up joins by converting predicates on small dimension tables to filters on large fact tables. This is useful when performing a join of multiple dimensions with one large fact table. The dimension keys on fact tables have lots of repeat values. The scan performance and repeat value optimization speeds up joins.

See Also: "Using Vector Joins to Enhance Join Performance" on page 2-14

Enhanced Aggregation Using VECTOR GROUP BY Transformations

An important aspect of analytics is to determine patterns and trends by aggregating data. Aggregations and complex SQL queries run faster when data is stored in the IM column store.

`VECTOR GROUP BY` transformations enable efficient in-memory array-based aggregation. During a fact table scan, aggregate values are accumulated into in-memory arrays and efficient algorithms are used perform aggregation. Performing joins based on the primary key and foreign key relationships are optimized for both star schemas and snowflake schemas.

See Also: "In-Memory Aggregation" on page 19-22

Using the Oracle In-Memory Column Store

You can store data using columnar format in the IM column store for existing databases or for new database that you plan to create. IM column store is simple to configure and does not impact existing applications. Depending on the requirement, you can configure one or more tablespaces, tables, materialized views, or partitions to be stored in memory.

> **See Also:** *Oracle Database Administrator's Guide* for information about configuring the IM column store

To store data in the IM column store:

1. Configure the `INMEMORY_SIZE` initialization parameter to specify the amount of memory that must be assigned to the IM column store.

   ```
   INMEMORY_SIZE = 100 GB
   ```

   > **See Also:** *Oracle Database Reference* for more information about the `INMEMORY_SIZE` parameter

2. Specify the database objects that must be stored in memory. Objects can include tablespaces, tables, materialized views, or partitions. Any queries on these objects will run faster than when the objects are stored on disk.

   For existing tablespaces, tables, or table partitions, use the `ALTER` command to store them in memory.

   ```
   ALTER TABLESPACE tbs1 INMEMORY;
   ALTER TABLE my_table MODIFY PARTITION p1 INMEMORY;
   ```

   While creating new tablespaces or tables, use the `INMEMORY` clause to specify that these objects must be stored in memory.

   ```
   CREATE TABLE my_table (id NUMBER, tname VARCHAR2(45)) INMEMORY;
   ```

   > **See Also:** *Oracle Database Administrator's Guide* for information about enabling objects to be stored in memory

3. Drop the indexes that were created to aid OLTP application workloads. Replace these with in-memory indexes. OLTP operations run faster because the objects that need to be accessed are now stored in memory.

   > **See Also:** *Oracle Database Administrator's Guide*

Using Vector Joins to Enhance Join Performance

Joins are an integral part of data warehousing workloads. IM column store enhances the performance of joins when the tables being joined are stored in memory. Simple joins that use bloom filters and complex joins between multiple tables benefit by using the IM column store. In a data warehousing environment, the most frequently-used joins are ones in which one or more dimension tables are joined with a fact table.

The following types of joins run faster when the tables being joined are stored in the IM column store:

- Joins that are amenable to using bloom filters
- Joins of multiple small dimension tables with one fact table
- Joins between two tables that have a PK-FK relationship

The IM column store runs queries that contain joins more efficiently and quickly by using vector joins. Vector joins allow the Oracle Database to take advantage of the fast scanning and vector processing capability of the IM column store. A vector join transforms a join between a dimension and fact table to filter that can be applied as part of the scan of the fact table. This join conversion is performed with the use of

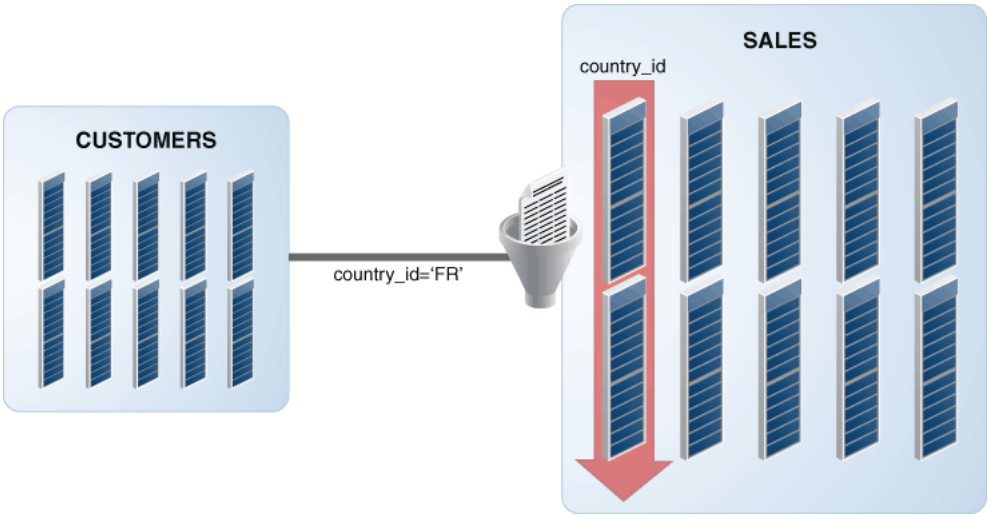
bloom filters, which enhance hash join performance in the Oracle Database. Although bloom filters are independent of IM column store, they can be applied very efficiently to data stored in memory through SIMD vector processing.

Consider the following query that performs a join of the CUSTOMERS dimension table with the SALES fact table:

```
SELECT c.customer_id, s.quantity_sold, s.amount_sold
FROM CUSTOMERS c, SALES s
WHERE c.customer_id = s.customer_id AND c.country_id = 'FR';
```

When both these tables are stored in the IM column store, SIMD vector processing is used to quickly scan the data and apply filters. Figure 2–4 displays a graphical representation of the how vector joins are used to implement the query. The predicate on the CUSTOMERS table, c.country_id='FR' is converted into a filter on the SALES fact table. The filter is country_id='FR'. Because the SALES table is stored in memory using columnar format, just one column needs to be scanned to determine the result of this query.

Figure 2–4 Vector Joins Using Oracle In-Memory Column Store



Automatic Big Table Caching to Improve the Performance of In-Memory Parallel Queries

Automatic big table caching enhances the in-memory query capabilities of Oracle Database. When a table does not fit in memory, the database decides which buffers to cache based on access patterns. This provides efficient caching for large tables, even if they do not fully fit in the buffer cache.

An optional section of the buffer cache, called the big table cache, is used to store data for table scans. The big table cache is integrated with the buffer cache and uses a temperature-based, object-level replacement algorithm to manage the big table cache contents. This is different from the access-based, block level LRU algorithm used by the buffer cache.

Note: The automatic big table caching feature is available starting with Oracle Database 12*c* Release 1 (12.1.0.2).

Typical data warehousing workloads scan multiple tables. Performance may be impacted if the combined size of these tables is greater than the combined size of the buffer cache. With automatic big table caching, the scanned tables are stored in the big table cache instead of the buffer cache. The temperature-based, object-level replacement algorithm used by the big table cache can provide enhanced performance for data warehousing workloads by:

- Selectively caching the "hot" objects

  Each time an object is accessed, Oracle Database increments the temperature of that object. An object in the big table cache can be replaced only by another object whose temperature is higher than its own temperature.

- Avoiding thrashing

  Partial objects are cached when objects cannot be fully cached.

In Oracle Real Application Clusters (Oracle RAC) environments, automatic big table caching is supported only for parallel queries. In single instance environments, this functionality is supported for both serial and parallel queries.

To use automatic big table caching, you must enable the big table cache. To use automatic big table caching for serial queries, you must set the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter to a nonzero value. To use automatic big table caching for parallel queries, you must set `PARALLEL_DEGREE_POLICY` to `AUTO` or `ADAPTIVE` and `DB_BIG_TABLE_CACHE_PERCENT_TARGET` to a nonzero value.

> **See Also:** *Oracle Database VLDB and Partitioning Guide* for more information about the big table cache and how it can be used

About In-Memory Aggregation

In-memory aggregation uses the `VECTOR GROUP BY` operation to enhance the performance of queries that aggregate data and join one or more relatively small tables to a larger table, as often occurs in a star query. `VECTOR GROUP BY` will be chosen by the SQL optimizer based on cost estimates. This will occur more often when the query selects from in-memory columnar tables and the tables include unique or numeric join keys (regardless of whether the uniqueness is forced by a primary key, unique constraint or schema design).

> **See Also:** *Oracle Database SQL Tuning Guide* for details about how In-memory aggregation works

`VECTOR GROUP BY` aggregation will only by chosen for `GROUP BY`. It will not be chosen for `GROUP BY ROLLUP`, `GROUPING SETS` or `CUBE`.

Note: This feature is available starting with Oracle Database 12*c* Release 1 (12.1.0.2).

This section contains the following topics:

- VECTOR GROUP BY Aggregation and the Oracle In-Memory Column Store
- When to Use VECTOR GROUP BY Aggregation
- When Is VECTOR GROUP BY Aggregation Used to Process Analytic Queries?

VECTOR GROUP BY Aggregation and the Oracle In-Memory Column Store

Although using the IM column store is not a requirement for using `VECTOR GROUP BY` aggregation, it is strongly recommended that you use both features together. Storing tables in memory using columnar format provides the foundation storage that `VECTOR GROUP BY` aggregation leverages to provide transactionally consistent results immediately after a schema is updated without the need to wait until the data marts are populated.

When to Use VECTOR GROUP BY Aggregation

Not all queries and scenarios benefit from the use of `VECTOR GROUP BY` aggregation. The following sections provide guidelines about the situations in which using this aggregation can be beneficial:

- Situations Where VECTOR GROUP BY Aggregation Is Useful
- Situations Where VECTOR GROUP BY Aggregation Is Not Advantageous

Situations Where VECTOR GROUP BY Aggregation Is Useful

`VECTOR GROUP BY` aggregation provides benefits in the following scenarios:

- The schema contains "mostly" unique keys or numeric keys for the columns that are used to join the fact and dimensions. The uniqueness can be enforced using a primary key, unique constraint or by schema design.
- The fact table is at least 10 times larger than the dimensions.
- The IM column store is used to store the dimensions and fact table in memory.

Situations Where VECTOR GROUP BY Aggregation Is Not Advantageous

Using `VECTOR GROUP BY` aggregation does not provide substantial performance benefits in the following scenarios:

- Joins are performed between two very large tables

  By default, the `VECTOR GROUP BY` transformation is used only if the fact table is at least 10 times larger than the dimensions.

- Dimensions contain more than 2 billion rows

  The `VECTOR GROUP BY` transformation is not used if a dimension contains more than 2 billion rows.

- The system does not have sufficient memory resources

  Most systems that use the IM column store will be able to benefit from using the `VECTOR GROUP BY` transformation.

When Is VECTOR GROUP BY Aggregation Used to Process Analytic Queries?

`VECTOR GROUP BY` aggregation is integrated with the Oracle Optimizer and no new SQL or initialization parameters are required to enable the use of this transformation. It also does not need additional indexes, foreign keys, or dimensions.

By default, Oracle Database decides whether or not to use `VECTOR GROUP BY` aggregation for a query based on the cost, relative to other execution plans that are determined for this query. However, you can direct the database to use `VECTOR GROUP BY` aggregation for a query by using query block hints or table hints.

VECTOR GROUP BY aggregation can be used to process a query that uses a fact view that is derived from multiple fact tables.

Oracle Database uses VECTOR GROUP BY aggregation to perform data aggregation when the following conditions are met:

- The queries or subqueries aggregate data from a fact table and join the fact table to one or more dimensions.

  Multiple fact tables joined to the same dimensions are also supported assuming that these fact tables are connected only through joins to the dimension. In this case, VECTOR GROUP BY aggregates fact table separately and then joins the results on the grouping keys.

- The dimensions and fact table are connected to each other only through join columns.

  Specifically, the query must not have any other predicates that refer to columns across multiple dimensions or from both a dimension and the fact table. If a query performs a join between two or more tables and then joins the result to the fact, then VECTOR GROUP BY aggregation treats the multiple dimensions as a single dimension.

The best performance for VECTOR GROUP BY aggregation is obtained when the tables being joined are stored in the IM column store.

VECTOR GROUP BY aggregation does not support the following:

- Semi- and anti-joins across multiple dimensions or between a dimension and the fact table
- Equi-joins across multiple dimensions.
- Aggregations performed using DISTINCT
- Bloom filters

  VECTOR GROUP BY aggregation and bloom filters are mutually exclusive.

  If bloom filters are used to perform joins for a query, then VECTOR GROUP BY aggregation is not applicable to the processing of this query.

3

Data Warehousing Physical Design

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 a tool such as 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:

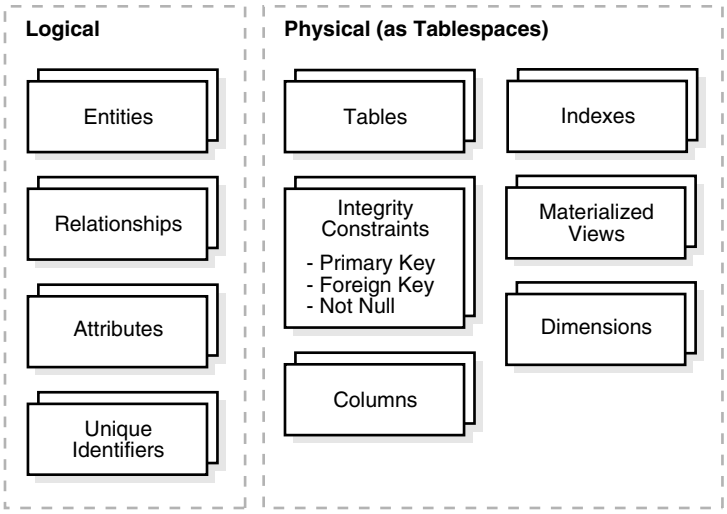
- *Oracle Database VLDB and Partitioning Guide* for further information regarding partitioning
- *Oracle Database Concepts* for further conceptual material regarding design matters.

About 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

This section contains the following topics:

Physical Design Structures

You must create some or all of the following structures as part of its physical design:

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.

About Partitioning

Oracle partitioning is an extremely important functionality for data warehousing, improving manageability, performance and availability. This section presents the key concepts and benefits of partitioning noting special value for data warehousing.

Partitioning allows tables, indexes or index-organized tables to be subdivided into smaller pieces. Each piece of the database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually. This gives the administrator considerable flexibility in managing a partitioned object. However, from the perspective of the user, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands.

Database objects - tables, indexes, and index-organized tables - are partitioned using a partitioning key, a set of columns that determine in which partition a given row will reside. For example a sales table partitioned on sales date, using a monthly partitioning strategy; the table appears to any application as a single, normal table. However, the DBA can manage and store each monthly partition individually, potentially using different storage tiers, applying table compression to the older data, or store complete ranges of older data in read only tablespaces.

Basic Partitioning Strategies Oracle partitioning offers three fundamental data distribution methods that control how the data is actually placed into the various individual partitions, namely:

- Range

  The data is distributed based on a range of values of the partitioning key (for a date column as the partitioning key, the 'January-2012' partition contains rows with the partitioning key values between '01-JAN-2012' and '31-JAN-2012'). The data distribution is a continuum without any holes and the lower boundary of a range is automatically defined by the upper boundary of the preceding range.

- List

  The data distribution is defined by a list of values of the partitioning key (for a region column as the partitioning key, the `North_America` partition may contain values `Canada`, `USA`, and `Mexico`). A special `DEFAULT` partition can be defined to catch all values for a partition key that are not explicitly defined by any of the lists.

- Hash

A hash algorithm is applied to the partitioning key to determine the partition for a given row. Unlike the other two data distribution methods, hash does not provide any logical mapping between the data and any partition.

Along with these fundamental approaches Oracle Database provides several more:

- Interval Partitioning

  An extension to range partitioning that enhances manageability. Partitions are defined by an interval, providing equi-width ranges. With the exception of the first partition all partitions are automatically created on-demand when matching data arrives.

- Partitioning by Reference

  Partitioning for a child table is inherited from the parent table through a primary key - foreign key relationship. Partition maintenance is simplified and partition-wise joins enabled.

- Virtual column based Partitioning

  Defined by one of the above mentioned partition techniques and the partitioning key is based on a virtual column. Virtual columns are not stored on disk and only exist as metadata. This approach enables a more flexible and comprehensive match of the business requirements.

Using the above-mentioned data distribution methods, a table can be partitioned either as single or composite partitioned table:

- Single (one-level) Partitioning

  A table is defined by specifying one of the data distribution methodologies, using one or more columns as the partitioning key. For example consider a table with a number column as the partitioning key and two partitions `less_than_five_hundred` and `less_than_thousand`, the `less_than_thousand` partition contains rows where the following condition is true: `500 <= Partitioning key <1000`.

  You can specify range, list, and hash partitioned tables.

- Composite Partitioning
- Combinations of two data distribution methods are used to define a composite partitioned table. First, the table is partitioned by data distribution method one and then each partition is further subdivided into subpartitions using a second data distribution method. All sub-partitions for a given partition together represent a logical subset of the data. For example, a range-hash composite partitioned table is first range-partitioned, and then each individual range-partition is further subpartitioned using the hash partitioning technique.

  See Also:

  - *Oracle Database VLDB and Partitioning Guide*
  - *Oracle Database Concepts* for more information about Hybrid Columnar Compression

Index Partitioning

Irrespective of the chosen index partitioning strategy, an index is either coupled or uncoupled with the partitioning strategy of the underlying table. The appropriate index partitioning strategy is chosen based on the business requirements, making partitioning well suited to support any kind of application. Oracle Database 12*c* differentiates between three types of partitioned indexes.

- Local Indexes

  A local index is an index on a partitioned table that is coupled with the underlying partitioned table, 'inheriting' the partitioning strategy from the table. Consequently, each partition of a local index corresponds to one - and only one - partition of the underlying table. The coupling enables optimized partition maintenance; for example, when a table partition is dropped, Oracle Database simply has to drop the corresponding index partition as well. No costly index maintenance is required. Local indexes are most common in data warehousing environments.

- Global Partitioned Indexes

  A global partitioned index is an index on a partitioned or nonpartitioned table that is partitioned using a different partitioning-key or partitioning strategy than the table. Global-partitioned indexes can be partitioned using range or hash partitioning and are uncoupled from the underlying table. For example, a table could be range-partitioned by month and have twelve partitions, while an index on that table could be hash-partitioned using a different partitioning key and have a different number of partitions. Global partitioned indexes are more common for OLTP than for data warehousing environments.

- Global Non-Partitioned Indexes

  A global non-partitioned index is essentially identical to an index on a non-partitioned table. The index structure is not partitioned and uncoupled from the underlying table. In data warehousing environments, the most common usage of global non-partitioned indexes is to enforce primary key constraints.

Partitioning for Manageability

A typical usage of partitioning for manageability is to support a 'rolling window' load process in a data warehouse. Suppose that a DBA loads new data into a table on a daily basis. That table could be range partitioned so that each partition contains one day of data. The load process is simply the addition of a new partition. Adding a single partition is much more efficient than modifying the entire table, because the DBA does not need to modify any other partitions. Another advantage of using partitioning is when it is time to remove data. In this situation, an entire partition can be dropped, which is very efficient and fast, compared to deleting each row individually.

Partitioning for Performance

By limiting the amount of data to be examined or operated on, partitioning provides a number of performance benefits. Two features specially worth noting are:

- Partitioning pruning: Partitioning pruning is the simplest and also the most substantial means to improve performance using partitioning. Partition pruning can often improve query performance by several orders of magnitude. For example, suppose an application contains an `ORDERS` table containing an historical record of orders, and that this table has been partitioned by day. A query requesting orders for a single week would only access seven partitions of the `ORDERS` table. If the table had two years of historical data, this query would access seven partitions instead of 730 partitions. This query could potentially execute 100x faster simply because of partition pruning. Partition pruning works with all of Oracle's other performance features. Oracle Database will utilize partition pruning in conjunction with any indexing technique, join technique, or parallel access method.

- Partition-wise joins: Partitioning can also improve the performance of multi-table joins, by using a technique known as partition-wise joins. Partition-wise joins can be applied when two tables are being joined together, and at least one of these tables is partitioned on the join key. Partition-wise joins break a large join into smaller joins of 'identical' data sets for the joined tables. 'Identical' here is defined as covering exactly the same set of partitioning key values on both sides of the join, thus ensuring that only a join of these 'identical' data sets will produce a result and that other data sets do not have to be considered. Oracle Database is using either the fact of already (physical) equi-partitioned tables for the join or is transparently redistributing ("repartitioning") one table at runtime to create equipartitioned data sets matching the partitioning of the other table, completing the overall join in less time. This offers significant performance benefits both for serial and parallel execution.

Partitioning for Availability

Partitioned database objects provide partition independence. This characteristic of partition independence can be an important part of a high-availability strategy. For example, if one partition of a partitioned table is unavailable, all of the other partitions of the table remain online and available. The application can continue to execute queries and transactions against this partitioned table, and these database operations will run successfully if they do not need to access the unavailable partition. The database administrator can specify that each partition be stored in a separate tablespace; this would allow the administrator to do backup and recovery operations on an individual partition or sets of partitions (by virtue of the partition-to-tablespace mapping), independent of the other partitions in the table. Therefore in the event of a disaster, the database could be recovered with just the partitions comprising the active data, and then the inactive data in the other partitions could be recovered at a convenient time, thus decreasing the system down-time.

In light of the manageability, performance and availability benefits, it should be part of every data warehouse.

See Also: *Oracle Database VLDB and Partitioning Guide*

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: *Oracle Database Concepts*

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: *Oracle Database Concepts*

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 5, "Basic Materialized Views"

Dimensions

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.

A dimension schema object 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 object 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.

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.

This section contains the following topics:

- Hierarchies
- Typical Dimension Hierarchy

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 can have complicated structures.

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.

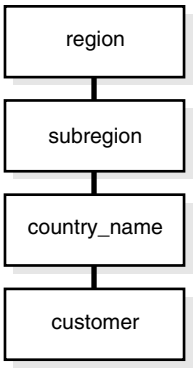
Level Relationships Level relationships specify top-to-bottom ordering of levels from most general (the root) to most specific information. They define the parent-child relationship between the levels in a hierarchy.

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 3–2 illustrates a dimension hierarchy based on `customers`.

Figure 3–2 Typical Levels in a Dimension Hierarchy



4

Data Warehousing Optimizations and Techniques

The following topics provide information about schemas in a data warehouse:

- Using Indexes in Data Warehouses
- Using Integrity Constraints in a Data Warehouse
- About Parallel Execution in Data Warehouses
- Optimizing Storage Requirements
- Optimizing Star Queries and 3NF Schemas

Using Indexes in Data Warehouses

This section discusses the following aspects of using indexes in data warehouses:

- Using Bitmap Indexes in Data Warehouses
- Benefits for Data Warehousing Applications
- Using B-Tree Indexes in Data Warehouses
- Using Index Compression
- Choosing Between Local Indexes and Global Indexes

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.

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.

This section contains the following topics:

- Cardinality
- Using Bitmap Join Indexes in Data Warehouses

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. This ratio is referred to 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 4–1 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 4–1 illustrates the bitmap index for the cust_gender column in this example. It consists of two separate bitmaps, one for gender.

Table 4–1 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 4–1. 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 4–1 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 4–2 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 SQL Language Reference*
- *Oracle Database VLDB and Partitioning Guide*

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, because 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. The large table is referred to as the fact table and the smaller tables as dimension tables. The following section describes the four different join models supported by bitmap join indexes.

Example 4–3 Bitmap Join Index: One Dimension Table Columns Joins One Fact Table

Unlike the example in "Bitmap Index" on page 4-3, where a bitmap index on the `cust_gender` column on the `customers` table was built, you 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 4–2 illustrates the bitmap representation for the bitmap join index in this example.

Table 4–2 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 4–4 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 4–5 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 4–6 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. In addition to key compression, OLTP index compression may provide a higher degree of compression, but is more appropriate for OLTP applications than data warehousing environments.

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 Database 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.

See Also:

- *Oracle Database Administrator's Guide* for more information regarding key compression
- *Oracle Database Administrator's Guide* for more information regarding OLTP index compression

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. Global indexes can be maintained without Oracle marking them as unusable after DDL, which makes global indexes 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.

Using Integrity Constraints 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 14, "Data Movement/ETL Overview"

This section contains the following topics:

- Overview of Constraint States
- Typical Data Warehouse Integrity Constraints

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 7, "Refreshing Materialized Views".
- 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 10, "Basic Query Rewrite for Materialized Views" 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 Also:

- Chapter 11, "Advanced Query Rewrite for Materialized Views" 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 7, "Refreshing Materialized Views" 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 5, "Basic Materialized Views"
- Chapter 10, "Basic Query Rewrite for Materialized Views"

About Parallel Execution in Data Warehouses

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.

This section contains the following topics:

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 VLDB and Partitioning Guide* for more information about using parallel execution

This section contains the following topics:

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, accessing or manipulating small sets of records or even single records, that characterize OLTP applications would not see any benefit from being executed in parallel.

When Not to Implement Parallel Execution

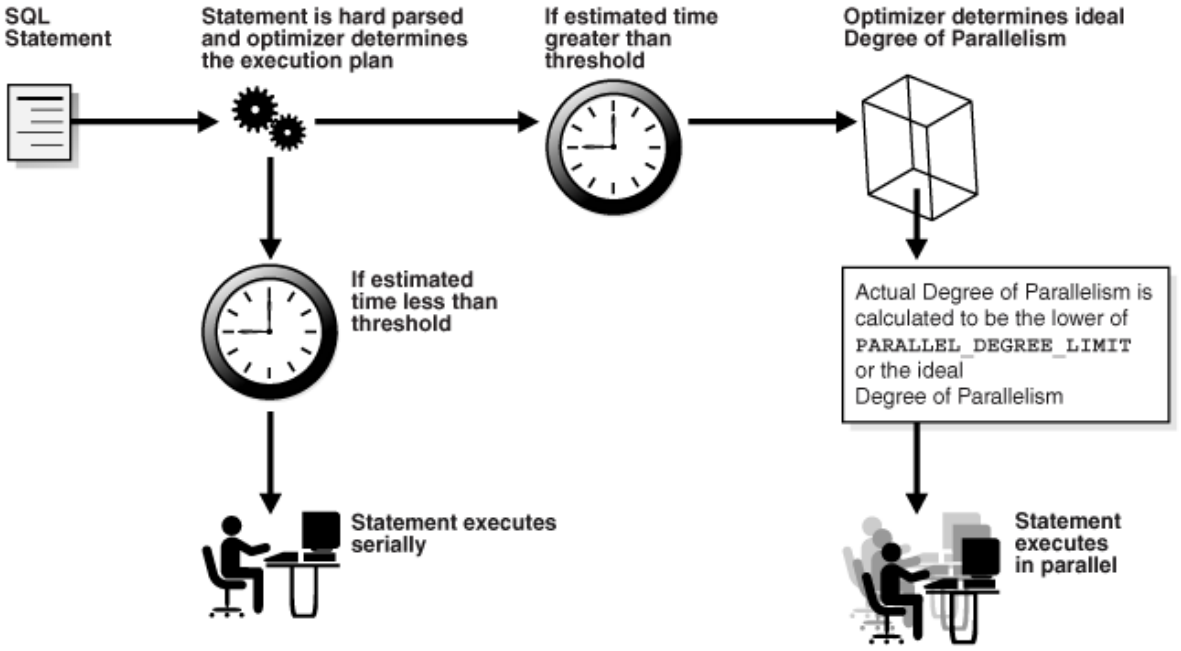
Parallel execution is not normally useful for:

- 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, even with parallel execution. 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 execution cost - the resource consumption of CPU, I/O, and memory - as determined by the Optimizer. That means that the database parses a query, calculates the cost and then determines a DOP to run with. The cheapest plan may be to run serially, which is also an option. Figure 4–2, "Optimizer Calculation: Serial or Parallel?" illustrates this decision making process.

Figure 4–2 Optimizer Calculation: Serial or Parallel?



Should you choose to use automatic DOP, you may potentially 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 must 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 workload 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 to exploit the aggregated database buffer cache of an Oracle Real Application Clusters environment, enables caching of objects in the terabyte range.

In-memory parallel execution takes advantage of this large aggregated database buffer cache. Having parallel execution servers accessing objects using the buffer cache enables full parallel in-memory processing of large volumes of data, leading to performance improvements in orders of magnitudes.

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

Optimizing Storage Requirements

You can reduce your storage requirements by compressing data, which is achieved by eliminating duplicate values in a database block. "Using Data Compression to Improve Storage" on page 4-17 describes how you can use compress data.

Database objects that can be compressed include tables and materialized views. For partitioned tables, you can compress some or all partitions. Compression attributes can be declared for a tablespace, a table, or a partition of a table. If declared at the tablespace level, then all tables created in that tablespace are compressed by default. You can alter the compression attribute for a table (or a partition or tablespace), and the change applies only to new data going into that table. As a result, a single table or partition may contain some compressed blocks and some regular blocks. This guarantees that data size will not increase as a result of compression. In cases where compression could increase the size of a block, it is not applied to that block.

Using Data Compression to Improve Storage

You can compress several partitions or a complete partitioned heap-organized table. You do this either by defining a complete partitioned table as being compressed, or by defining it on a per-partition level. Partitions without a specific declaration inherit the attribute from the table definition or, if nothing is specified on the table level, from the tablespace definition.

The decision about whether or not a partition should be compressed is based on the same rules as a nonpartitioned table. Because of the ability of range and composite partitioning to separate data logically into distinct partitions, a partitioned table is an ideal candidate for compressing parts of the data (partitions) that are mainly read-only. It is, for example, beneficial in all rolling window operations as a kind of intermediate stage before aging out old data. With data compression, you can keep more old data online, minimizing the burden of additional storage use.

You can also change any existing uncompressed table partition later, add new compressed and uncompressed partitions, or change the compression attribute as part

of any partition maintenance operation that requires data movement, such as MERGE PARTITION, SPLIT PARTITION, or MOVE PARTITION. The partitions can contain data, or they can be empty.

The access and maintenance of a partially or fully compressed partitioned table are the same as for a fully uncompressed partitioned table. All rules that apply to fully uncompressed partitioned tables are also valid for partially or fully compressed partitioned tables.

To use data compression:

The following example creates a range-partitioned table with one compressed partition costs_old. The compression attribute for the table and all other partitions is inherited from the tablespace level.

```
CREATE TABLE costs_demo (
   prod_id     NUMBER(6),    time_id     DATE,
   unit_cost   NUMBER(10,2), unit_price  NUMBER(10,2))
PARTITION BY RANGE (time_id)
   (PARTITION costs_old
       VALUES LESS THAN (TO_DATE('01-JAN-2003', 'DD-MON-YYYY')) COMPRESS,
    PARTITION costs_q1_2003
       VALUES LESS THAN (TO_DATE('01-APR-2003', 'DD-MON-YYYY')),
    PARTITION costs_q2_2003
       VALUES LESS THAN (TO_DATE('01-JUN-2003', 'DD-MON-YYYY')),
    PARTITION costs_recent VALUES LESS THAN (MAXVALUE));
```

Optimizing Star Queries and 3NF Schemas

Oracle data warehouses can work well with star schemas and third normal form schemas. This section presents important techniques for optimizing performance in both types of schema. For conceptual background on star and 3NF schemas, see "About Third Normal Form Schemas" on page 2-2. and "About Star Schemas" on page 2-5.

You should consider the following when using star queries:

- Optimizing Star Queries
- Using Star Transformation
- Optimizing Third Normal Form Schemas

Optimizing Star Queries

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. "Tuning Star Queries" on page 4-18 describes how to improve the performance of star queries.

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.

This section contains the following topics:

- Star Transformation with a Bitmap Index
- Execution Plan for a Star Transformation with a Bitmap Index
- Star Transformation with a Bitmap Join Index
- Execution Plan for a Star Transformation with a Bitmap Join Index
- How Oracle Chooses to Use Star Transformation
- Star Transformation Restrictions

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 4-19:

```
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 4-22:

```
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

Optimizing Third Normal Form Schemas

Optimizing a third normal form (3NF) schema requires the following:

- Power

  Power means that the hardware configuration must be balanced. Many data warehouse operations are based upon large table scans and other IO-intensive operations, which perform vast quantities of random IOs. In order to achieve optimal performance the hardware configuration must be sized end to end to sustain this level of throughput. This type of hardware configuration is called a balanced system. In a balanced system, all components - from the CPU to the disks - are orchestrated to work together to guarantee the maximum possible IO throughput.

- Partitioning

  The larger tables should be partitioned using composite partitioning (range-hash or list-hash). There are three reasons for this:

  – Easier manageability of terabytes of data

  – Faster accessibility to the necessary data

  – Efficient and performant table joins

- Parallel Execution

  Parallel Execution enables a database task to be parallelized or divided into smaller units of work, thus allowing multiple processes to work concurrently. By using parallelism, a terabyte of data can be scanned and processed in minutes or less, not hours or days.

The rest of this section on 3NF optimization discusses partitioning, with special attention to partition-wise joins, followed by parallel query execution.

This section contains the following topics:

3NF Schemas: Partitioning

Partitioning allows a table, index or index-organized table to be subdivided into smaller pieces. Each piece of the database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics. From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually.

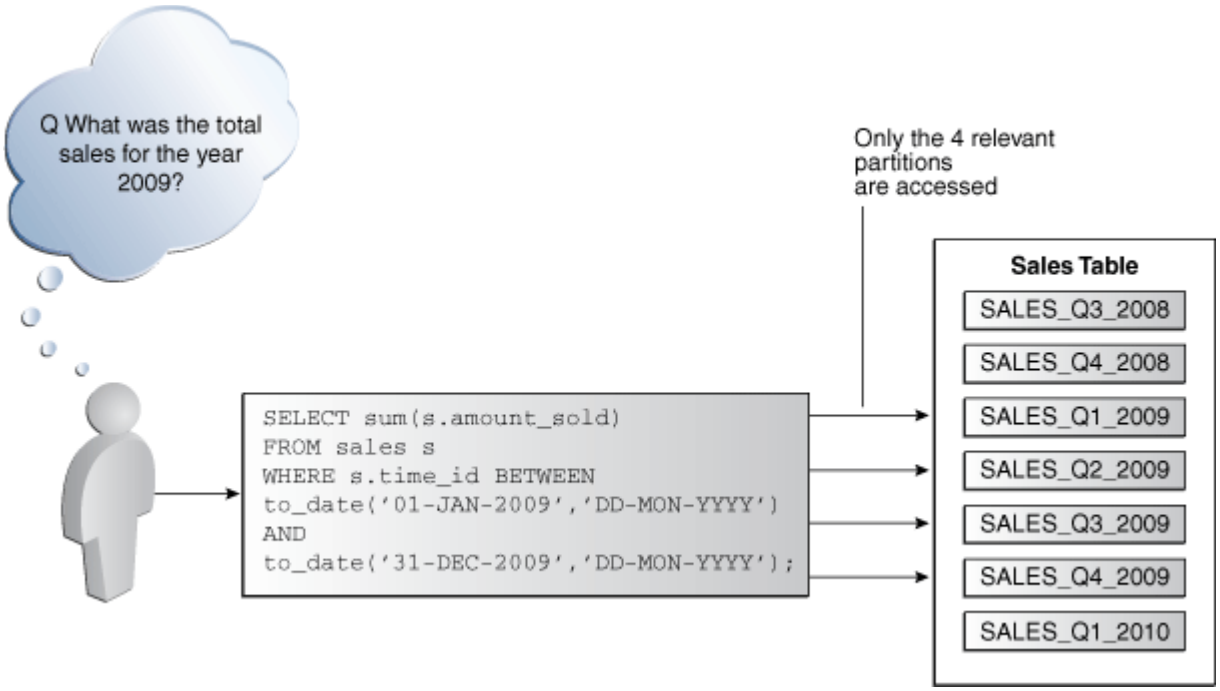
This gives the administrator considerable flexibility in managing partitioned objects. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table; no modifications are necessary when accessing a partitioned table using SQL DML commands. Partitioning can provide tremendous benefits to a wide variety of applications by improving manageability, availability, and performance.

Partitioning for Manageability Range partitioning will help improve the manageability and availability of large volumes of data. Consider the case where two year's worth of sales data or 100 terabytes (TB) is stored in a table. At the end of each day a new batch of data needs to be to loaded into the table and the oldest days worth of data needs to be removed. If the Sales table is ranged partitioned by day the new data can be loaded using a partition exchange load. This is a sub-second operation and should have little or no impact to end user queries. In order to remove the oldest day of data simply issue the following command:

```
SH@DBM1 > ALTER TABLE SALES DROP PARTITION Sales_Q4_2009;
```

Partitioning for Easier Data Access Range partitioning will also help ensure only the necessary data to answer a query will be scanned. Let's assume that the business users predominately accesses the sales data on a weekly basis, e.g. total sales per week then range partitioning this table by day will ensure that the data is accessed in the most efficient manner, as only 4 partitions need to be scanned to answer the business users query instead of the entire table. The ability to avoid scanning irrelevant partitions is known as partition pruning.

Figure 4–3 Partition Pruning



Partitioning for Join Performance Sub-partitioning by hash is used predominately for performance reasons. Oracle uses a linear hashing algorithm to create sub-partitions. In order to ensure that the data gets evenly distributed among the hash partitions it is highly recommended that the number of hash partitions is a power of 2 (for example, 2, 4, 8, and so on). Each hash partition should be at least 16MB in size. Any smaller and they will not have efficient scan rates with parallel query.

One of the main performance benefits of hash partitioning is partition-wise joins. Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. This significantly reduces response time and improves both CPU and memory resource usage. In a clustered data warehouse, this significantly reduces response times by limiting the data traffic over the interconnect (IPC), which is the key to achieving good scalability for massive join operations. Partition-wise joins can be full or partial, depending on the partitioning scheme of the tables to be joined.

A full partition-wise join divides a join between two large tables into multiple smaller joins. Each smaller join performs a joins on a pair of partitions, one for each of the tables being joined. For the optimizer to choose the full partition-wise join method, both tables must be equi-partitioned on their join keys. That is, they have to be partitioned on the same column with the same partitioning method. Parallel execution of a full partition-wise join is similar to its serial execution, except that instead of joining one partition pair at a time, multiple partition pairs are joined in parallel by multiple parallel query servers. The number of partitions joined in parallel is determined by the Degree of Parallelism (DOP).

Figure 4–4 Full Partition-Wise Join

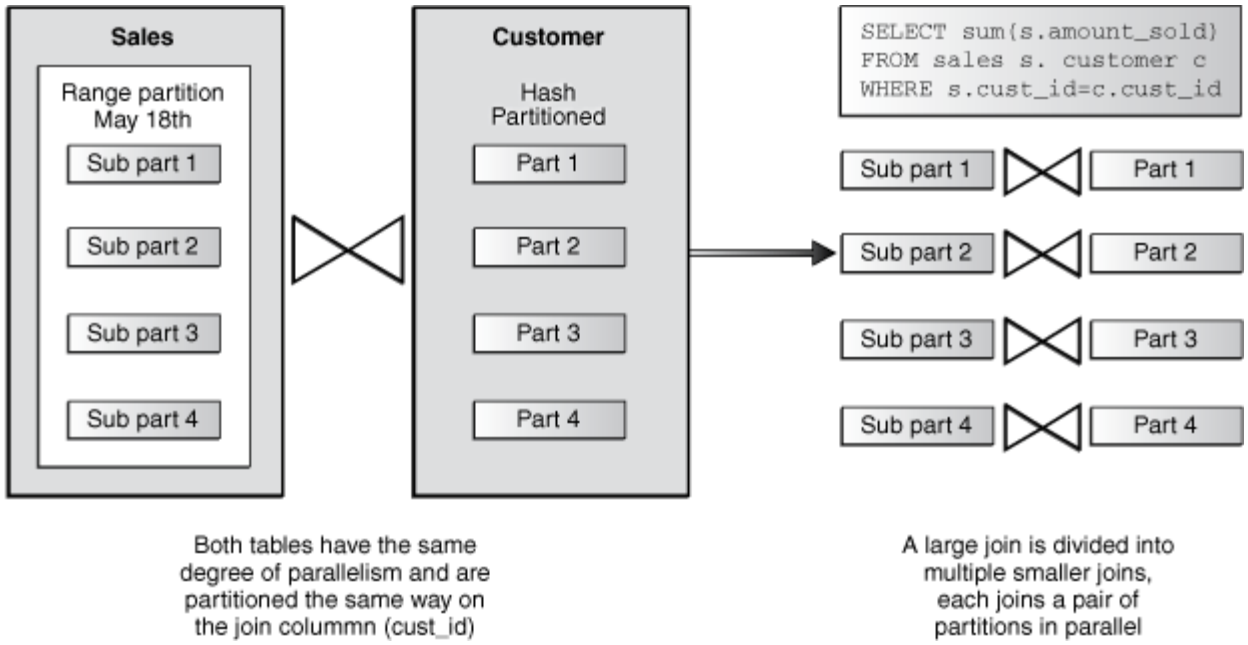


Figure 4–4 illustrates the parallel execution of a full partition-wise join between two tables, `Sales` and `Customers`. Both tables have the same degree of parallelism and the same number of partitions. They are range partitioned on a date field and sub partitioned by hash on the `cust_id` field. As illustrated in the picture, each partition pair is read from the database and joined directly. There is no data redistribution necessary, thus minimizing IPC communication, especially across nodes. Figure 4–5 below shows the execution plan you would see for this join.

To ensure that you get optimal performance when executing a partition-wise join in parallel, the number of partitions in each of the tables should be larger than the degree of parallelism used for the join. If there are more partitions than parallel servers, each parallel server will be given one pair of partitions to join, when the parallel server completes that join, it will requests another pair of partitions to join. This process repeats until all pairs have been processed. This method enables the load to be balanced dynamically (for example, 128 partitions with a degree of parallelism of 32).

What happens if only one of the tables you are joining is partitioned? In this case the optimizer could pick a partial partition-wise join. Unlike full partition-wise joins, partial partition-wise joins can be applied if only one table is partitioned on the join key. Hence, partial partition-wise joins are more common than full partition-wise joins. To execute a partial partition-wise join, Oracle dynamically repartitions the other table based on the partitioning strategy of the partitioned table. Once the other table is repartitioned, the execution is similar to a full partition-wise join. The redistribution operation involves exchanging rows between parallel execution servers. This operation leads to interconnect traffic in Oracle RAC environments, because data needs to be repartitioned across node boundaries.

Figure 4–5 Partial Partition-Wise Join

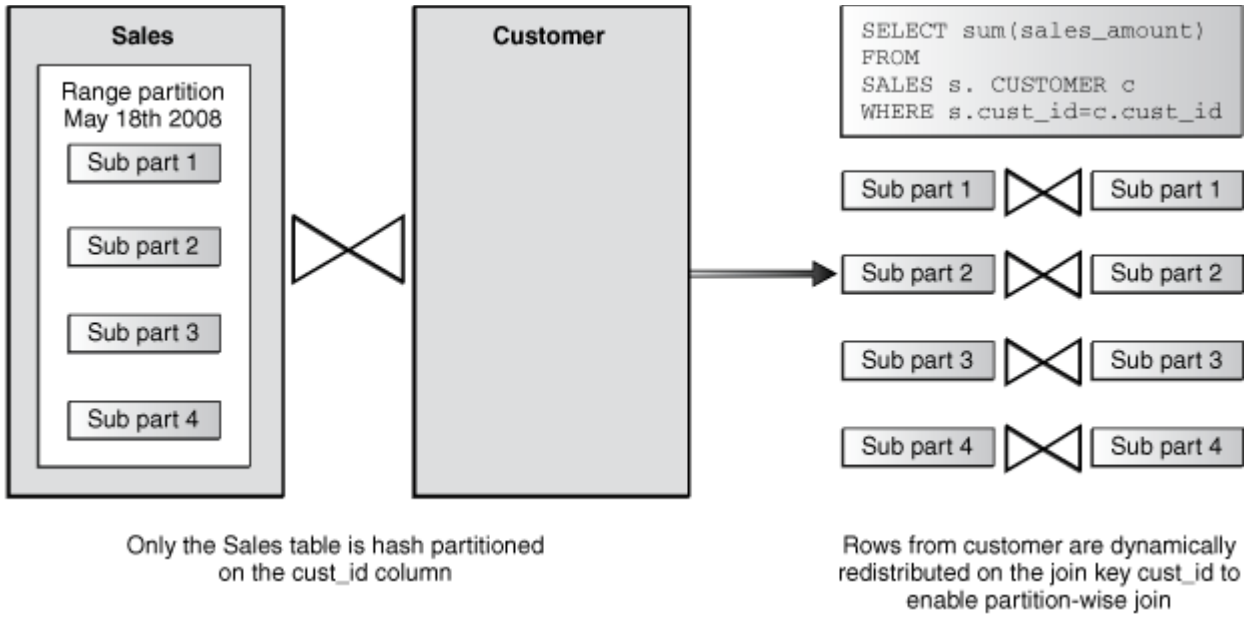


Figure 4–5 illustrates a partial partition-wise join. It uses the same example as in Figure 4–4, except that the customer table is not partitioned. Before the join operation is executed, the rows from the customers table are dynamically redistributed on the join key.

3NF Schemas: Parallel Query Execution

3NF schemas can leverage parallelism in multiple ways, but here the focus is on one facet of parallelism that is specially significant to 3NF: SQL parallel execution for large queries. SQL parallel execution in the Oracle Database is based on the principles of a coordinator (often called the Query Coordinator or QC) and parallel servers. The QC is the session that initiates the parallel SQL statement and the parallel servers are the individual sessions that perform work in parallel. The QC distributes the work to the parallel servers and may have to perform a minimal mostly logistical - portion of the work that cannot be executed in parallel. For example a parallel query with a `SUM()` operation requires adding the individual sub-totals calculated by each parallel server.

The QC is easily identified in the parallel execution in Figure 4–5 as PX COORDINATOR. The process acting as the QC of a parallel SQL operation is the actual user session process itself. The parallel servers are taken from a pool of globally available parallel server processes and assigned to a given operation. The parallel servers do all the work shown in a parallel plan BELOW the QC.

By default, the Oracle Database is configured to support parallel execution out-of-the-box and is controlled by two initialization parameters `parallel_max_servers` and `parallel_min_servers`. While parallel execution provides a very powerful and scalable framework to speed up SQL operations, you should not forget to use some common sense rules; while parallel execution might buy you an additional incremental performance boost, it requires more resources and might also have side effects on other users or operations on the same system. Small tables/indexes (up to thousands of records; up to 10s of data blocks) should never be enabled for parallel execution. Operations that only hit small tables will not benefit much from executing in parallel, but they will use parallel servers that you will want to be available for operations accessing large tables. Remember also that once an operation starts at a certain degree of parallelism (DOP), there is no way to reduce its

DOP during the execution. The general rules of thumb for determining the appropriate DOP for an object are:

- Objects smaller than 200 MB should not use any parallelism
- Objects between 200 MB and 5GB should use a DOP of 4
- Objects beyond 5GB use a DOP of 32

Needless to say the optimal settings may vary on your system- either in size range or DOP - and highly depend on your target workload, the business requirements, and your hardware configuration.

Whether or Not to Use Cross Instance Parallel Execution in Oracle RAC By default, Oracle Database enables inter-node parallel execution (parallel execution of a single statement involving more than one node). As mentioned earlier, the interconnect in an Oracle RAC environment must be sized appropriately as inter-node parallel execution may result in a lot of interconnect traffic. If you are using a relatively weak interconnect in comparison to the I/O bandwidth from the server to the storage subsystem, you may be better off restricting parallel execution to a single node or to a limited number of nodes. Inter-node parallel execution will not scale with an undersized interconnect. From Oracle Database 11*g* onwards, it is recommended to use Oracle RAC services to control parallel execution on a cluster.

Optimizing Star Queries Using VECTOR GROUP BY Aggregation

`VECTOR GROUP BY` aggregation optimizes queries that aggregate data and join one or more relatively small tables to a larger table. This transformation can be chosen by the SQL optimizer based on cost estimates. In the context of data warehousing, `VECTOR GROUP BY` will often be chosen for star queries that select data from in-memory columnar tables.

`VECTOR GROUP BY` aggregation is similar to a bloom filter in that it transforms the join condition between a small table and a large table into a filter on the larger table. `VECTOR GROUP BY` aggregation further enhances query performance by aggregating data during the scan of the fact table rather than as a separate step following the scan.

See Also:

- "About In-Memory Aggregation" on page 2-16
- *Oracle Database SQL Tuning Guide* for a detailed `VECTOR GROUP BY` scenario

Part II

Optimizing Data Warehouses

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

It contains the following chapters:

5

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 hundreds of gigabytes to a few terabytes. Usually, the vast majority of the data is stored in a few very large **fact table**s.

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.

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.

This section contains the following topics:

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 Also:

- *Oracle Database Heterogeneous Connectivity User's Guide*
- *Oracle Database Advanced Replication*

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 Also:

- *Oracle Database Heterogeneous Connectivity User's Guide*
- *Oracle Database Advanced Replication*

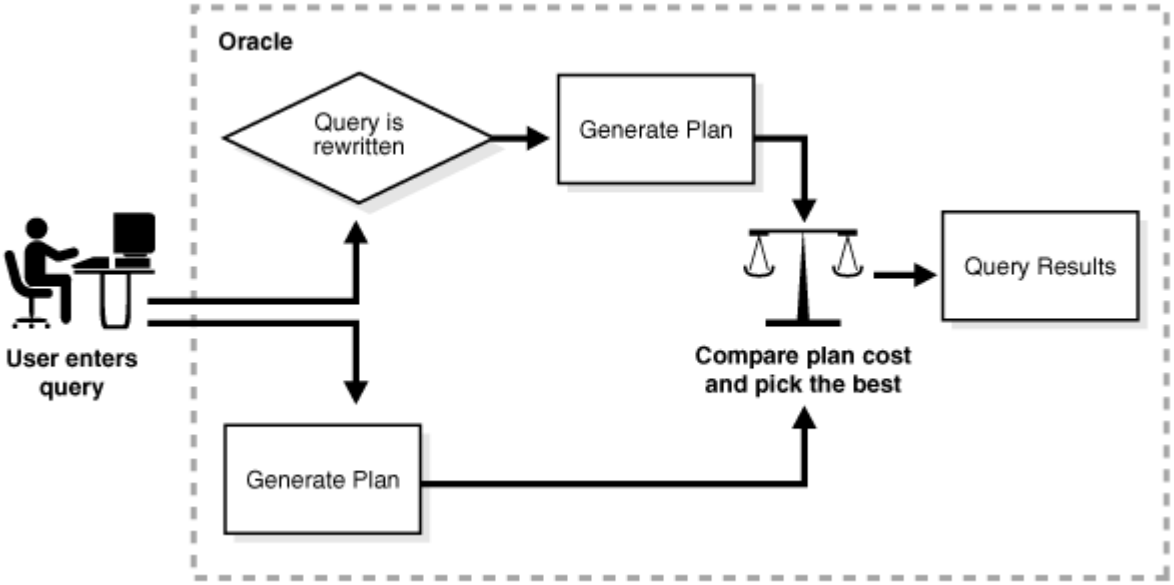
The Need for Materialized Views

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 5–1 illustrates how query rewrite works.

Figure 5–1 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 Database 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 depends. 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 Also: *Oracle Database Advanced Replication*

Components of Summary Management

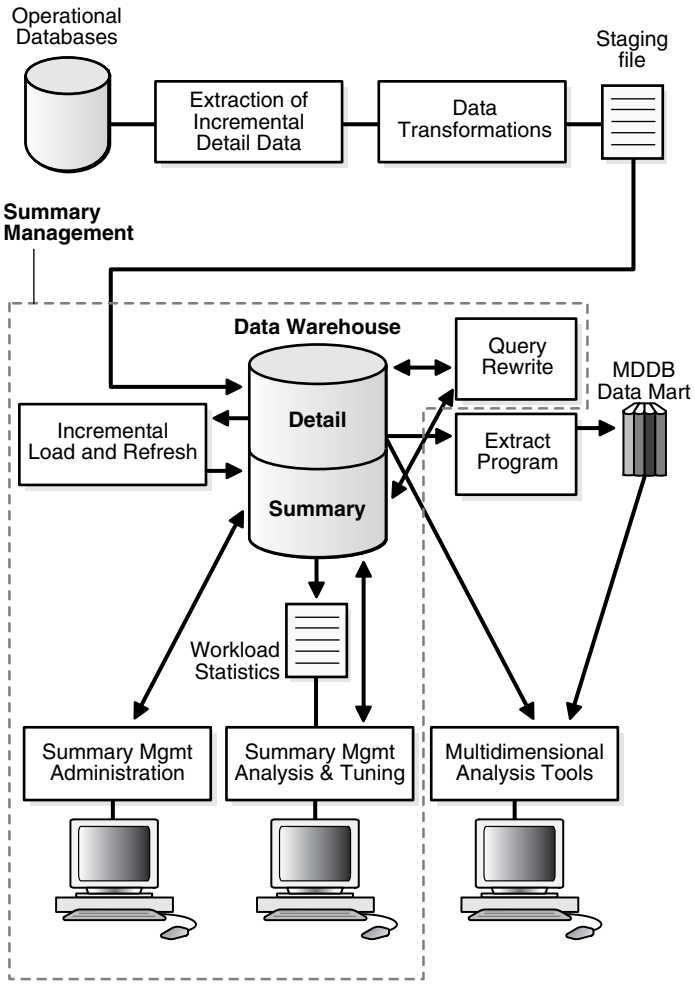
Summary management consists of:

- Mechanisms to define **materialized view**s and **dimension**s.
- 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.
- The `TUNE_MVIEW` package, 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 5–2 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 5–2 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 9, "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.

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.

This section contains the following topics:

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.

Guidelines for Materialized View Schema Design

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.

Dimensions Guideline 1 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.

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.

Dimensions Guideline 2 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.

Dimensions Guideline 3 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.

Dimensions Guideline 4 After each load and before refreshing your materialized view, use the VALIDATE_DIMENSION procedure of the DBMS_DIMENSION package to incrementally verify dimensional integrity.

Incremental Loads Guideline 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.

Partitions Guideline Range/composite partition your tables by a monotonically increasing time column if possible (preferably of type DATE).

Time Dimensions Guideline 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.

See Also:

- "Types of Materialized Views" on page 5-9
- "Creating Dimensions" on page 9-3 for details on the benefits of maintaining a child-side row join with a parent-side row
- *Oracle Database SQL Language Reference*

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 5–1. 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 log**s 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 5-23.

> **See Also:** "Requirements for Using Materialized Views with Aggregates" on page 5-12

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 5–1 Creating a Materialized View (Total Number and Value of Sales)

```
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 5–2 Creating a Materialized View (Computed Sum of Sales)

```
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 5–3 Creating a Materialized View (Aggregates on a Single Table)

```
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.

> **See Also:** *Oracle Database SQL Language Reference* for syntax of the CREATE MATERIALIZED VIEW and CREATE MATERIALIZED VIEW LOG statements

Requirements for Using Materialized Views with Aggregates

Table 5–1 illustrates the aggregate requirements for materialized views. If aggregate X is present, aggregate Y is required and aggregate Z is optional.

Table 5–1 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 5–4 on page 5-13, 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.

> **See Also:** "Materialized Join Views FROM Clause Considerations" on page 5-13

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 Database 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 partition change tracking (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 Also:

- "Restrictions on Fast Refresh on Materialized Views with Joins Only" on page 5-23 for more information regarding the conditions that cause refresh performance to degrade.
- "Partition Change Tracking (PCT) Refresh" on page 7-4

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 5–4 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.

This section contains the following topics:

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 5–5 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 5-15. 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 5–6 illustrates creating a materialized view called cust_sales_mv.

Example 5–6 Example 4: 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 5-29.

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

   This section contains the following topics:

   - Creating Materialized Views with Column Alias Lists
   - Materialized Views Names
   - Storage And Table CompressionRefresh Options
   - Build Methods
   - Enabling Query Rewrite
   - Query Rewrite Restrictions

- ORDER BY Clause
- Using Oracle Enterprise Manager
- Using Materialized Views with NLS Parameters
- Adding Comments to Materialized Views
- Using the FORCE Option With Materialized View Logs

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.

Materialized Views Names

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.

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

Build Methods

Two build methods are available for creating the materialized view, as shown in Table 5–2. 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.

See Also: Chapter 7, "Refreshing Materialized Views"

Table 5–2 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
- General Query Rewrite Restrictions

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 data types 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.
- The defining query cannot contain any references to objects or XMLTYPEs.
- A materialized view is a noneditioned object and cannot depend on editioned objects unless it mentions an evaluation edition in which names of editioned objects are to be resolved.
- A materialized view may only be eligible for query rewrite in a specific range of editions. The *query_rewrite_clause* in the CREATE or ALTER MATERIALIZED VIEW

statement lets you specify the range of editions in which a materialized view is eligible for query rewrite.

See Also:

- Chapter 11, "Advanced Query Rewrite for Materialized Views"
- *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 11, "Advanced Query Rewrite for Materialized Views"
- *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 5–3 describes the refresh modes.

Table 5–3 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 5–4 describes the refresh options.

Table 5–4 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 5–5 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.

This section contains the following topics:

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 5-22.
- 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 5-22.

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 5–1 on page 5-12 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 5–1 on page 5-12 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.

  The max/min fast refresh after delete or mixed DML does not have the same behavior as the insert-only case. It deletes and recomputes the max/min values for the affected groups. You need to be aware of its performance impact.
- 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, see *Oracle Database SQL 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 11-46 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 Materialized Views on Prebuilt Tables For materialized views created with the prebuilt option, the index `I_snap$` is not created by default. This index helps fast refresh performance, and a description of how to create this index is illustrated in "Choosing Indexes for Materialized Views" on page 5-30.

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 5–7 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.

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 comments to materialized views.

Example: Adding Comments to a Materialized View

The following statement adds a comment to data dictionary views for an existing materialized view:

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

To view the comment after the preceding statement execution, you 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 you drop the materialized view, the comment will be passed to the prebuilt table with the comment: '(from materialized view) Sales summary data'.

Creating Materialized View Logs

Materialized view logs are required if you want to use fast refresh, with the exception of partition change tracking refresh. That is, if a detail table supports partition change tracking 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;
  ```

This section contains the following topics:

- Using the FORCE Option With Materialized View Logs
- Materialized View Log Purging

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

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 data type. 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 Also:** Chapter 10, "Basic Query Rewrite for Materialized Views" 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.

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 you should create a new materialized view that does include the time dimension (if possible). Also, in this case, 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.

If you create a materialized view with the prebuilt option, the I_snap$ index is not automatically created. This index significantly improves fast refresh performance, and you can create it manually by issuing a statement such as the following:

```
CREATE UNIQUE INDEX <OWNER>."I_SNAP$_<MVIEW_NAME>" ON <OWNER>.<MVIEW_NAME>
 (SYS_OP_MAP_NONNULL("LOG_DATE"))
  PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DE
FAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE <TABLESPACE_NAME>;
```

> **See Also:** *Oracle Database SQL 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 partition change tracking refresh is possible

Using this procedure is straightforward and described in "Using the DBMS_MVIEW.EXPLAIN_MVIEW Procedure" on page 5-32. 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 5–6 on page 5-35, and all the possible messages are listed in Table 5–7 on page 5-36.

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

This section contains the following topics:

- DBMS_MVIEW.EXPLAIN_MVIEW Declarations
- Using MV_CAPABILITIES_TABLE
- MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details
- MV_CAPABILITIES_TABLE Column Details

DBMS_MVIEW.EXPLAIN_MVIEW Declarations

The following PL/SQL declarations that are made for you in the DBMS_MVIEW package show the order and data types 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 5-6
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 5–8 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 7, "Refreshing Materialized Views" for further details about partition change tracking
- Chapter 11, "Advanced Query Rewrite for Materialized Views" for further details about partition change tracking

MV_CAPABILITIES_TABLE.CAPABILITY_NAME Details

Table 5–6 lists explanations for values in the CAPABILITY_NAME column.

Table 5–6 CAPABILITY_NAME Column Details

CAPABILITY_NAME	Description
PCT	If this capability is possible, partition change tracking is possible on at least one detail relation. If this capability is not possible, partition change tracking 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.
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, partition change tracking (PCT) applies to the partitioned table named in the RELATED_TEXT column. PCT is needed to support fast refresh 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.

Table 5–6 (Cont.) CAPABILITY_NAME Column Details

CAPABILITY_NAME	Description
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 5–7 lists the semantics for RELATED_TEXT and RELATED_NUM columns.

Table 5–7 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

Table 5–7 (Cont.) MV_CAPABILITIES_TABLE Column Details

MSGNO	MSGTXT	RELATED_NUM	RELATED_TEXT
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
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		

6

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 Also:** *Oracle Database VLDB and Partitioning Guide* for further details about partitioning

This section contains the following topics:

About Partition Change Tracking

It is possible and advantageous to track freshness to a finer grain than the entire materialized view. You can achieve this through partition change tracking (PCT), which is a method to identify which rows in a materialized view are affected by a certain detail table partition. 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 expressions in the materialized view.

> **See Also:** "Join Dependent Expression" on page 6-3 for more information

Note that, while partition change tracking tracks the staleness on a partition and subpartition level (for composite partitioned tables), the level of granularity for PCT refresh is only the top-level partitioning strategy. Consequently, any change to data in one of the subpartitions of a composite partitioned-table will only mark the single impacted subpartition as stale and have the rest of the table available for rewrite, but the PCT refresh will refresh the whole partition that contains the impacted subpartition.

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 with range or list as the top-level partitioning strategy.
- 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.
- 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.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for details regarding the `DBMS_MVIEW.PMARKER` function and partition markers

This section contains the following topics:

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 6–1 Materialized View with Partition Key

```
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 because 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 6–2 Creating a Materialized View: 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 (the ratio of distinct values to the number of table rows) of the materialized view (see Example 6–3 for an example). The function returns a partition identifier that uniquely identifies the partition or subpartition for a specified row within a specified partitioned 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 6–3 Using Partition Markers in a Materialized View

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. "Benefits of Partitioning a Materialized View" on page 6-6 describes the benefits of partitioning a prebuilt table. The following example illustrates this:

```
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 "About Partition

Change Tracking" on page 6-1.

- 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 Also:** Chapter 7, "Refreshing Materialized Views" 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.

This section contains the following topics:

- Materialized Views and Hierarchical Cubes
- Benefits of Partitioning Materialized Views
- Compressing Materialized Views
- Materialized Views with Set Operators

Materialized Views and Hierarchical 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 6–4 Hierarchical Cube

Consider a sales data set with two dimensions, each of which has a four-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 6–1 shows the four levels of each dimension.

Table 6–1 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 Also:

- *Oracle Database SQL Language Reference* for data compression syntax and restrictions
- "Storage And Table Compression" on page 5-18 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 Also: "Restrictions on Fast Refresh on Materialized Views with UNION ALL" on page 5-24 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 6–5 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 6–6 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 21, "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 or READ 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 or READ 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 the `READ WITH GRANT` or `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` or `READ` 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.

> **See Also:** Querying Materialized Views with Virtual Private Database (VPD)

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, because security processing of the underlying base relations is performed against the owner of the materialized view.

This section contains the following topics:

- Using Query Rewrite with Virtual Private Database
- Restrictions with Materialized Views and Virtual Private Database

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 6-11

7

Refreshing Materialized Views

This chapter discusses how to refresh materialized views, which is a key element in maintaining good performance and consistent data when working with materialized views in a data warehousing environment.

This chapter includes the following sections:

Refreshing Materialized Views

The database maintains data in materialized views by refreshing them after changes to the base tables. The refresh method can be incremental or a complete refresh. There are two incremental refresh methods, known as log-based refresh and partition change tracking (PCT) refresh. The incremental refresh is commonly called `FAST` refresh as it usually performs faster than the complete refresh.

A complete refresh occurs when the materialized view is initially created when it is defined as `BUILD IMMEDIATE`, unless the materialized view references a prebuilt table or is defined as `BUILD DEFERRED`. Users can perform a complete refresh at any time after the materialized view is created. The complete refresh involves executing the query that defines the materialized view. This process can be slow, especially if the database must read and process huge amounts of data.

An incremental refresh eliminates the need to rebuild materialized views from scratch. Thus, processing only the changes can result in a very fast refresh time. Materialized views can be refreshed either on demand or at regular time intervals. Alternatively, materialized views in the same database as their base tables can be refreshed whenever a transaction commits its changes to the base tables.

For materialized views that use the log-based fast refresh method, a materialized view log and/or a direct loader log keep a record of changes to the base tables. A materialized view log is a schema object that records changes to a base table so that a materialized view defined on the base table can be refreshed incrementally. Each materialized view log is associated with a single base table. The materialized view log resides in the same database and schema as its base table.

The PCT refresh method can be used if the modified base tables are partitioned and the modified base table partitions can be used to identify the affected partitions or portions of data in the materialized view. When there have been some partition maintenance operations on the base tables, this is the only incremental refresh method

that can be used. The PCT refresh removes all data in the affected materialized view partitions or affected portions of data and recomputes them from scratch.

For each of these refresh methods, you have two options for how the refresh is performed, namely in-place refresh and out-of-place refresh. The in-place refresh executes the refresh statements directly on the materialized view. The out-of-place refresh creates one or more outside tables and executes the refresh statements on the outside tables and then switches the materialized view or affected materialized view partitions with the outside tables. Both in-place refresh and out-of-place refresh achieve good performance in certain refresh scenarios. However, the out-of-place refresh enables high materialized view availability during refresh, especially when refresh statements take a long time to finish.

Also adopting the out-of-place mechanism, a new refresh method called synchronous refresh is introduced in Oracle Database 12*c*, Release 1. It targets the common usage scenario in the data warehouse where both fact tables and their materialized views are partitioned in the same way or their partitions are related by a functional dependency.

The refresh approach enables you to keep a set of tables and the materialized views defined on them to be always in sync. In this refresh method, the user does not directly modify the contents of the base tables but must use the APIs provided by the synchronous refresh package that will apply these changes to the base tables and materialized views at the same time to ensure their consistency. The synchronous refresh method is well-suited for data warehouses, where the loading of incremental data is tightly controlled and occurs at periodic intervals.

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 the case of `ON DEMAND` materialized views, the refresh can be performed with refresh methods provided in either the `DBMS_SYNC_REFRESH` or the `DBMS_MVIEW` packages:

- The `DBMS_SYNC_REFRESH` package contains the APIs for synchronous refresh, a new refresh method introduced in Oracle Database 12*c*, Release 1. For details, see Chapter 8, "Synchronous Refresh".
- The `DBMS_MVIEW` package contains the APIs whose usage is described in this chapter. There are three basic types of refresh operations: complete refresh, fast refresh, and partition change tracking (PCT) refresh. These basic types have been enhanced in Oracle Database 12*c*, Release 1 with a new refresh option called out-of-place refresh.

The `DBMS_MVIEW` package contains three APIs for performing 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 7-6 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 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.

An additional option when performing refresh is to use out-of-place refresh, where outside tables are used to improve materialized view availability and refresh performance in certain situations.

See Also:

- *Oracle OLAP User's Guide* for information regarding the refresh of cube organized materialized views
- "The Out-of-Place Refresh Option" on page 7-4 for a discussion of out-of-place refresh

This section contains the following topics:

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 5-6, 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 "About Partition Change Tracking" on page 6-1 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 6-6 and hence, make the PCT refresh process more efficient.

See Also:

- "About Partition Change Tracking" on page 6-1 for more information regarding partition change tracking

The Out-of-Place Refresh Option

Beginning with Oracle Database 12*c* Release 1, a new refresh option is available to improve materialized view refresh performance and availability. This refresh option is called out-of-place refresh because it uses outside tables during refresh as opposed to the existing "in-place" refresh that directly applies changes to the materialized view container table. The out-of-place refresh option works with all existing refresh methods, such as `FAST` (`'F'`), `COMPLETE` (`'C'`), `PCT` (`'P'`), and `FORCE` (`'?'`). Out-of-place refresh is particularly effective when handling situations with large amounts of data

changes, where conventional DML statements do not scale well. It also enables you to achieve a very high degree of availability because the materialized views that are being refreshed can be used for direct access and query rewrite during the execution of refresh statements. In addition, it helps to avoid potential problems such as materialized view container tables becoming fragmented over time or intermediate refresh results being seen.

In out-of-place refresh, the entire or affected portions of a materialized view are computed into one or more outside tables. For partitioned materialized views, if partition level change tracking is possible, and there are local indexes defined on the materialized view, the out-of-place method also builds the same local indexes on the outside tables. This refresh process is completed by either switching between the materialized view and the outside table or partition exchange between the affected partitions and the outside tables. During refresh, the outside table is populated by direct load, which is efficient.

This section contains the following topics:

Types of Out-of-Place Refresh

There are three types of out-of-place refresh:

- out-of-place fast refresh

  This offers better availability than in-place fast refresh. It also offers better performance when changes affect a large part of the materialized view.

- out-of-place PCT refresh

  This offers better availability than in-place PCT refresh. There are two different approaches for partitioned and non-partitioned materialized views. If truncation and direct load are not feasible, you should use out-of-place refresh when the changes are relatively large. If truncation and direct load are feasible, in-place refresh is preferable in terms of performance. In terms of availability, out-of-place refresh is always preferable.

- out-of-place complete refresh

  This offers better availability than in-place complete refresh.

Using the refresh interface in the `DBMS_MVIEW` package, with `method = ?` and `out_of_place = true`, out-of-place fast refresh are attempted first, then out-of-place PCT refresh, and finally out-of-place complete refresh. An example is the following:

```
DBMS_MVIEW.REFRESH('CAL_MONTH_SALES_MV', method => '?',
   atomic_refresh => FALSE, out_of_place => TRUE);
```

Restrictions and Considerations with Out-of-Place Refresh

Out-of-place refresh has all the restrictions that apply when using the corresponding in-place refresh. In addition, it has the following restrictions:

- Only materialized join views and materialized aggregate views are allowed
- No `ON COMMIT` refresh is permitted
- No remote materialized views, cube materialized views, object materialized views are permitted
- No `LOB` columns are permitted

- Not permitted if materialized view logs, triggers, or constraints (except NOT NULL) are defined on the materialized view
- Not permitted if the materialized view contains the CLUSTERING clause
- Not applied to complete refresh within a CREATE or ALTER MATERIALIZED VIEW session or an ALTER TABLE session
- Atomic mode is not permitted. If you specify atomic_refresh as TRUE and out_of_place as TRUE, an error is displayed

For out-of-place PCT refresh, there is the following restriction:

- No UNION ALL or grouping sets are permitted

For out-of-place fast refresh, there are the following restrictions:

- No UNION ALL, grouping sets or outer joins are permitted
- Not allowed for materialized join views when more than one base table is modified with mixed DML statements

Out-of-place refresh requires additional storage for the outside table and the indexes for the duration of the refresh. Thus, you must have enough available tablespace or auto extend turned on.

The partition exchange in out-of-place PCT refresh impacts the global index on the materialized view. Therefore, if there are global indexes defined on the materialized view container table, Oracle disables the global indexes before doing the partition exchange and rebuild the global indexes after the partition exchange. This rebuilding is additional overhead.

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 7–1 details the refresh options.

Table 7–1 ON DEMAND Refresh Methods

Refresh Option	Parameter	Description
COMPLETE	C	Refreshes by recalculating the defining query of the materialized view.

Table 7–1 (Cont.) ON DEMAND Refresh Methods

Refresh Option	Parameter	Description
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 Advanced Replication* for information showing how to use it in a replication environment
- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the DBMS_MVIEW package

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 stops after it encounters the first error, and any remaining materialized views in the list are not 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 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.
- Whether to use out-of-place refresh

  This parameter works with all existing refresh methods (F, P, C, ?). So, for example, if you specify F and out_of_place = true, then an out-of-place fast refresh is attempted. Similarly, if you specify P and out_of_place = true, then out-of-place PCT refresh is attempted.

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, 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, 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.

- Whether to use out-of-place refresh

  This parameter works with all existing refresh method (F, P, C, ?). So, for example, if you specify F and out_of_place = true, then an out-of-place fast refresh is attempted. Similarly, if you specify P and out_of_place = true, then out-of-place PCT refresh is attempted.

An example of refreshing all materialized views is the following:

```
DBMS_MVIEW.REFRESH_ALL_MVIEWS(failures,'C','', TRUE, FALSE, 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 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.

- 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.

- Whether to use out-of-place refresh

  This parameter works with all existing refresh methods (F, P, C, ?). So, for example, if you specify F and out_of_place = true, then an out-of-place fast refresh is attempted. Similarly, if you specify P and out_of_place = true, then out-of-place PCT refresh is attempted.

To perform a full refresh on all materialized views that reference the customers table, specify:

```
DBMS_MVIEW.REFRESH_DEPENDENT(failures, 'CUSTOMERS', 'C', '', FALSE, 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 7-13 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 are requeued and continue running. To remove these jobs, use the `DBMS_JOB.REMOVE` procedure.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `DBMS_JOB` package

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.

If you are not sure how to make a materialized view fast refreshable, you can use the `DBMS_ADVISOR.TUNE_MVIEW` procedure, which provides a script containing the statements required to create a fast refreshable materialized view.

See Also:

- *Oracle Database SQL Tuning Guide*
- Chapter 5, "Basic Materialized Views" for further information about the `DBMS_MVIEW` package

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*

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_MVIEWS, 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

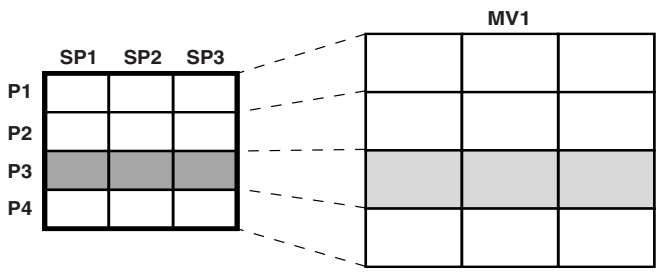
  To determine which partitions are fresh.

- *_USER_MVIEW_DETAIL_SUBPARTITION

  To determine which subpartitions are fresh.

The use of these views is illustrated in the following examples. Figure 7–1 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 7–1 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 7–1 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 7–2 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 7–3 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 7–4 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 19-26. 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

This section contains the following topics with tips on refreshing materialized views:

- Tips for Refreshing Materialized Views with Aggregates
- Tips for Refreshing Materialized Views Without Aggregates
- Tips for Refreshing Nested 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). Starting in Oracle Database 12*c*, the database automatically gathers table statistics as part of a bulk-load operation (CTAS and IAS) similar to how statistics are gathered when an index is created. By gathering statistics during the data load, you avoid additional scan operations and provide the necessary statistics as soon as the data becomes available to the users. Note that, in the case of an IAS statement, statistics are only gathered if the table the data is being inserted into is empty.

  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 is 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 is 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. "About Partition Change Tracking" on page 6-1 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 causes a 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 uses 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 is 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 enhances 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 greatly enhances 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 is 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 must refresh them in the correct order (taking into account the dependencies between the materialized views) because the nested materialized view are 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 is refreshed with respect to the current contents of the other materialized views (that is, the other materialized views are not 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 are 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 are 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 automatically uses 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) is 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" on page 7-19 provides additional information about PCT refresh.

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 occurs automatically and no user intervention is required in order for it to occur. See "About Partition Change Tracking" on page 6-1 for PCT requirements.

The following examples illustrate the use of this feature:

- PCT Fast Refresh Scenario 1
- PCT Fast Refresh Scenario 2
- PCT Fast Refresh Scenario 3

PCT Fast Refresh Scenario 1

In this scenario, 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.

1. Create the materialized view. 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. Run the DBMS_MVIEW.EXPLAIN_MVIEW procedure to determine which tables 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 allows 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 applies 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 5-31 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 automatically performs 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 this scenario, the first three steps are the same as in "PCT Fast Refresh Scenario 1" on page 7-19. 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 is 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, 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 is 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, FALSE);
```

Fast refresh will automatically detect that PCT is available and perform a PCT refresh.

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 proceeds 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, data types, 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, you must do two things. First, you must add a new partition to the `sales` table. You use an `ALTER TABLE ... ADD PARTITION` statement. This adds 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, you can add our newly created table to this partition using the `EXCHANGE PARTITION` operation. This exchanges 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 preserves 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 automatically maintains 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.

Note that, if you use synchronous refresh, instead of performing Step 3, you must register the sales_01_2001 table using the DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION package. See Chapter 8, "Synchronous Refresh" for more information.

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 four 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 is immediately 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 empties 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 16-2 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. You also assume that at least one compressed partition is already part of the partitioned table.

See Also:

- *Oracle Database Administrator's Guide* for more information regarding 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, you 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 is 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. You 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, you have to create one matching partition, having the range boundaries you 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 you 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 describes the following two typical scenarios where partitioning is used with refresh:

- Refresh Scenario 1

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 is 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

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 7–5 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 7–6 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 7–7 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 Database performs a regular join of the source and the target tables. When the UPDATE clause is omitted, Oracle Database performs an antijoin of the source and the target tables. This makes the join between the source and target table more efficient.

Example 7–8 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 7–9 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 only occurs 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 7–10 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 7–11 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 are optimized and the join condition are 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 do 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, 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 series of DELETE statements, and none of the data in the sales table needs to be moved. However, this approach also has some disadvantages. When removing a large percentage of rows, the DELETE statement leaves 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 is not 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.

8

Synchronous Refresh

This chapter describes a method to synchronize changes to the tables and materialized views in a data warehouse. This method is based on synchronizing updates to tables and materialized views, and is called synchronous refresh.

This chapter includes the following sections:

About Synchronous Refresh

Synchronous refresh is a refresh method introduced in Oracle Database 12*c* Release 1 that enables you to keep a set of tables and the materialized views defined on them to be always in sync. It is well-suited for data warehouses, where the loading of incremental data is tightly controlled and occurs at periodic intervals.

In most data warehouses, the fact tables are partitioned along the time dimension and, very often, the incremental data load consists mainly of changes to recent time periods. Synchronous refresh exploits these characteristics to greatly improve refresh performance and throughput. This results in fast query performance for both planned and ad hoc queries, which is key to a successful data warehouse.

This section describes the main requirements and basic concepts of synchronous refresh, and includes the following:

What Is Synchronous Refresh?

Synchronous refresh is a new approach for maintaining tables and materialized views in a data warehouse where tables and materialized views are refreshed at the same time. In traditional refresh methods, the changes are applied to the base tables and the materialized views are refreshed separately with one of the following refresh methods:

- Log-based incremental (fast) refresh using materialized view logs if such logs are available
- PCT refresh if it is applicable
- Complete refresh

Synchronous refresh combines some elements of log-based incremental (fast) refresh and PCT refresh methods, but it is applicable only to `ON DEMAND` materialized views, unlike the other two methods. There are three major differences between it and the other refresh methods:

- Synchronous refresh requires you to register the tables and materialized views.
- Synchronous refresh requires you to specify changes to the data according to some formally specified rules.
- Synchronous refresh works by dividing the refresh operation into two steps: preparation and execution. This approach provides some important advantages over the other methods, such as better performance and more control.

Synchronous refresh APIs are defined in a new package called `DBMS_SYNC_REFRESH`. For more information about this package, see *Oracle Database PL/SQL Packages and Types Reference*.

Why Use Synchronous Refresh?

Synchronous refresh offers the following advantages over traditional types of refresh methods:

- It coordinates the loading of the changes into the base tables with the extremely efficient refresh of the dependent materialized views themselves.
- It decreases the time materialized views are not available to be used by the Optimizer to rewrite queries.
- It is well-suited for a wide class of materialized views (materialized aggregate views and materialized join views) commonly used in data warehouses. It does require the materialized views be partitioned as well as the fact tables, and if materialized views are not currently partitioned, they can be efficiently partitioned to take advantage of synchronous refresh.
- It fully exploits partitioning and the nature of the data warehouse load cycle to guarantee synchronization between the materialized view and the base table throughout the refresh procedure.
- In a typical data warehouse, data preparation consists of extracting the data from one or more sources, cleansing, and formatting it for consistency, and transforming into the data warehouse schema. The data preparation area is called the staging area and the base tables in a data warehouse are loaded from the tables in the staging area. The synchronous refresh method fits into this model because it allows you to load change data into the staging logs.
- The staging logs play the same role as materialized view logs in the conventional fast refresh method. There is, however, an important difference. In the conventional fast refresh method, the base table is first updated and the changes

are then applied from the materialized view log to the materialized views. But in the synchronous refresh method, the changes from the staging log are applied to refresh the materialized views while also being applied to the base tables.

- Most materialized views in a data warehouse typically employ a star or snowflake schema with fact and dimension tables joined in a foreign key to primary key relationship. The synchronous refresh method can handle both schemas in all possible change data load scenarios, ranging from rows being added to only the fact table, to arbitrary changes to the fact and dimension tables.
- Instead of providing the change load data in the staging logs, you have a choice of directly providing the change data in the form of outside tables containing the data to be exchanged with the affected partition in the base table. This capability is provided by the `REGISTER_PARTITION_OPERATION` procedure in the `DBMS_SYNC_REFRESH` package.

Registering Tables and Materialized Views for Synchronous Refresh

Before actually performing synchronous refresh, you must register the appropriate tables and materialized views. Synchronous refresh provides these methods to register tables and materialized views:

- Tables are registered with synchronous refresh by creating a **staging log** on them. A staging log is created with the `CREATE MATERIALIZED VIEW LOG` statement whose syntax has been extended in this release to create staging logs as well as the familiar materialized view logs used for the traditional incremental refresh. After you create a staging log on a table, it is deemed to be registered with synchronous refresh and can be modified only by using the synchronous refresh procedures. In other words, a table with a staging log defined on it is registered with synchronous refresh and cannot be modified directly by the user.

  For more information about the `CREATE MATERIALIZED VIEW LOG` statement, see *Oracle Database SQL Language Reference*.

- Materialized views are registered with synchronous refresh using the `REGISTER_MVIEWS` procedure in the `DBMS_SYNC_REFRESH` package. The `REGISTER_MVIEWS` procedure implicitly creates groups of related objects called sync refresh groups. A **sync refresh group** consists of all related materialized views and tables that must be refreshed together as a single entity because they are dependent on one another.

  For more information about the `DBMS_SYNC_REFRESH` package, see *Oracle Database PL/SQL Packages and Types Reference*.

Specifying Change Data for Refresh

In the other refresh methods, you can directly modify the base tables of the materialized view, and the issue of specifying change data does not arise. But with synchronous refresh, you are required to specify and prepare the change data according to certain formally specified rules and using APIs provided by the `DBMS_SYNC_REFRESH` package.

There are two ways to specify the change data:

- Provide the change data in an outside table and register it with the `REGISTER_PARTITION_OPERATION` procedure.

  See "Working with Partition Operations" on page 8-12 for more details.

- Provide the change data by in staging logs and process them with the `PREPARE_STAGING_LOG` procedure. The format of the staging logs and rules for populating

are described in "Working with Staging Logs" on page 8-14. You are required to run the PREPARE_STAGING_LOG procedure for every table before performing the refresh operation on that table.

Synchronous Refresh Preparation and Execution

After preparing the change data, you can perform the actual refresh operation. Synchronous refresh takes a new approach to refresh execution. It works by dividing the refresh operation into two steps: preparation and execution. This is one of the main differences between it and the other refresh methods and provides some important benefits.

The preparation step determines the mapping between the fact table partitions and the materialized view partitions. This step computes the new tables corresponding only to the partitions of the fact table that have been changed by the incremental change data load. After these tables, called **outside tables**, have been computed, the actual execution of the refresh operation takes place in the execution step, which consists of just exchanging the outside tables with the corresponding partitions in the fact table or materialized view.

By dividing the refresh execution step into two phases and providing separate procedures for them, synchronous refresh not only provides you control over the refresh execution process, but also improves overall system performance. It does this by minimizing the time the materialized views are not available for use by direct access or the Optimizer because they are modified by the refresh process. During the preparation phase, the materialized view and its tables are not modified because at this time all the refresh changes are recorded in the outside table. Consequently, the materialized view is available to any query that needs to read them. It is only during execution that the tables and materialized views are modified. Execution performance is mainly affected by the number of changes to the dimension tables; if this number is small, then the performance should be very good because the exchange partition operations are themselves very fast.

The DBMS_SYNC_REFRESH package provides the PREPARE_REFRESH and EXECUTE_REFRESH procedures to perform these two steps.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference*

Materialized View Eligibility Rules and Restrictions for Synchronous Refresh

The primary requirement for a materialized view to be eligible for synchronous refresh is that the materialized view must be partitioned with a key that can be derived from the partition key of its fact table. The following sections describe the other requirements for eligibility for synchronous refresh.

This section contains the following topics:

- Synchronous Refresh Restrictions: Partitioning
- Synchronous Refresh Restrictions: Refresh Options
- Synchronous Refresh Restrictions: Constraints
- Synchronous Refresh Restrictions: Tables
- Synchronous Refresh Restrictions: Materialized Views
- Synchronous Refresh Restrictions: Materialized Views with Aggregates

Synchronous Refresh Restrictions: Partitioning

There are two key requirements to use synchronous refresh:

- The materialized view must be partitioned along the same dimension as the fact table.
- The partition key of the fact table should functionally determine the partition key of the materialized view.

The term *functionally determine* means the partition key of the materialized view can be derived from the partition key of the fact table based on a foreign key constraint relationship. This condition is satisfied if the partition key of the materialized view is the same as that for the fact table or related by joins from the fact table to the dimension table as in a star or snowflake schema. For example, if the fact table is partitioned by a date column, such as `TIME_KEY`, the materialized view can be partitioned by `TIME_KEY`, `MONTH`, or `YEAR`.

Synchronous refresh supports two types of partitioning on fact tables and materialized views: range partitioning and composite partitioning, when the top-level partitioning type is range.

Synchronous Refresh Restrictions: Refresh Options

When you define a materialized view, you can specify three refresh options: how to refresh; whether trusted constraints can be used; and what type of refresh is to be performed. If unspecified, the defaults are assumed to be `ON DEMAND`, `ENFORCED` constraints, and `FORCE` respectively. Synchronous refresh requires that the first two of these options must have the values `ON DEMAND` and `TRUSTED` constraints respectively. Synchronous refresh does not require the type of refresh to have any specific value, so it can be `FAST`, `FORCE`, or `COMPLETE`.

Synchronous Refresh Restrictions: Constraints

The relationships between the fact and dimension tables are declared by foreign and primary key constraints on the tables. Synchronous refresh trusts these constraints to perform the refresh, and requires that `USING TRUSTED CONSTRAINTS` must be specified in the materialized view definition. This allows using nonvalidated `RELY` constraints and rewriting against materialized views in an `UNKNOWN` or `FRESH` state during refresh.

When a table is registered for synchronous refresh, its constraints might be in a `VALIDATE` or `NOVALIDATE` state. If the table is a dimension table, synchronous refresh will retain this state during the refresh execution process.

However, if the table is a fact table, synchronous refresh marks the constraints `NOVALIDATE` state during refresh execution. This avoids the need for validating the constraint on existing data during a partition exchange that is the basis of the synchronous refresh method, and improves the performance of refresh execution.

Because the constraints on the fact table are not enforced by synchronous refresh, it is you who must verify the integrity and consistently of the data provided.

Synchronous Refresh Restrictions: Tables

To be eligible for synchronous refresh, a table must satisfy the following conditions:

- The table cannot have VPD or triggers defined on it.
- The table cannot have any `RAW` type.
- The table cannot be remote.

- The staging log key of each table registered for synchronous refresh should satisfy the requirements described in "Staging Log Key" on page 8-15.

Synchronous Refresh Restrictions: Materialized Views

There are some other restrictions that are specific to materialized views registered for synchronous refresh:

- The `ROWID` column cannot be used to define the query. It is not relevant because it uses partition exchange, which replaces the original partition with the outside table. Hence, the defining query should not include any `ROWID` columns.
- Synchronous refresh does not support nested materialized views, `UNION ALL` materialized views, subqueries, or complex queries in the materialized view definition. The defining query must conform to the star or snowflake schema.
- These SQL constructs are also not supported: analytic window functions (such as `RANK`), the `MODEL` clause, and the `CONNECT BY` clause.
- Synchronous refresh is not supported for a materialized view that refers to views, remote tables, or outer joins.
- The materialized view must not contain references to nonrepeating expressions like `SYSDATE` and `ROWNUM`.

In general, most restrictions that apply to PCT-refresh, fast refresh, and general query rewrite also apply to synchronous refresh. Those restrictions are available at:

- "Materialized View Restrictions" on page 5-19
- "General Query Rewrite Restrictions" on page 5-20
- "General Restrictions on Fast Refresh" on page 5-22

Synchronous Refresh Restrictions: Materialized Views with Aggregates

For materialized views with aggregates, synchronous refresh shares these restrictions with fast refresh:

- Only `SUM`, `COUNT`, `AVG`, `STDDEV`, `VARIANCE`, `MIN`, and `MAX` are supported.
- `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.
- If `VARIANCE(expr)` or `STDDEV(expr)` is specified, `COUNT(expr)` and `SUM(expr)` must be specified. Oracle recommends that `SUM(expr *expr)` be specified.

Using Synchronous Refresh

Synchronous refresh differs from the other refresh methods in a number of ways. One is that the API for synchronous refresh is contained in a new package called `DBMS_SYNC_REFRESH`, whereas other refresh methods are declared in the `DBMS_MVIEW` package. Another difference is that after objects are registered with synchronous refresh, and, once registered, the other refresh methods cannot be used with them.

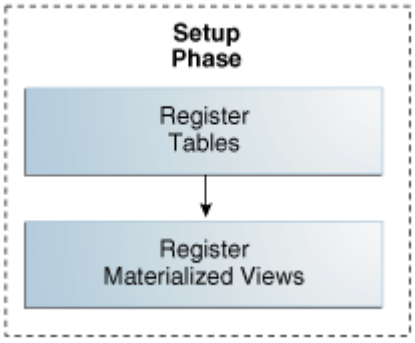
The operations associated with synchronous refresh can be divided into the following three broad phases:

- The Registration Phase

The Registration Phase

In this phase (Figure 8–1), you register the objects for use with synchronous refresh. The two steps in this phase are registration of tables first and then materialized views. You register the tables (by creating staging logs) and materialized views (with the `REGISTER_MVIEWS` procedure). The staging logs are created with the `CREATE MATERIALIZED LOG ... FOR SYNCHRONOUS REFRESH` statement. If a table already has a regular materialized view log, the `ALTER MATERIALIZED LOG ... FOR SYNCHRONOUS REFRESH` statement can be used to convert it to a staging log.

Figure 8–1 Registration Phase



You can create a staging log with a statement, as show in Example 8–1.

Example 8–1 Registering Tables

```
CREATE MATERIALIZED VIEW LOG ON fact
FOR SYNCHRONOUS REFRESH USING st_fact;
```

If a table has a materialized view log, you can alter it to a staging log with a statement, such as the following:

```
ALTER MATERIALIZED VIEW LOG ON fact
FOR SYNCHRONOUS REFRESH USING st_fact;
```

You can register a materialized view with a statement, as shown in Example 8–2.

Example 8–2 Registering Materialized Views

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('MV1');
```

You can register multiple materialized views at one time:

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('mv2, mv2_year, mv1_halfmonth');
```

The Synchronous Refresh Phase

Figure 8–2 shows the synchronous refresh phase. This phase can be used repeatedly to perform synchronous refresh. The three main steps in this phase are:

1. Prepare the change data for the refresh operation. You can provide the change data in a table and register it with the `REGISTER_PARTITION_OPERATION` procedure or provide the data by populating the staging logs. The staging logs must be

processed with the PREPARE_STAGING_LOG procedure before proceeding to the next step.

An example is Example 8–12.

2. Perform the first step of the refresh operation (PREPARE_REFRESH). This can potentially be a long-running operation because it prepares and loads the outside tables.

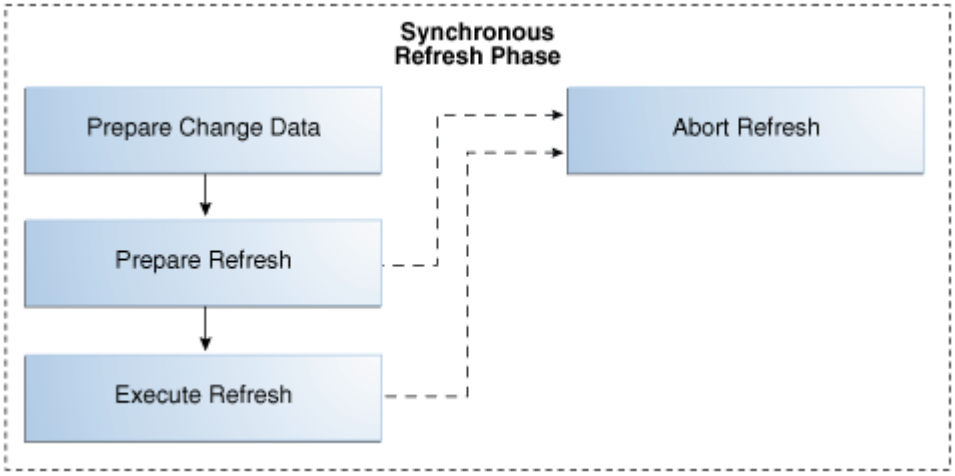
   An example is Example 8–16.

3. Perform the second and last step of the refresh operation (EXECUTE_REFRESH). This usually runs very fast because it usually consists of a series of partition-exchange operations.

   An example is Example 8–20.

In Figure 8–2, solid arrows show the standard control flow and dashed arrows are used for error-handling cases. If either of the refresh operations (PREPARE_REFRESH or EXECUTE_REFRESH) raises user errors, you use an ABORT_REFRESH procedure to restore tables and materialized views to the state that existed before the refresh operation, fix the problem, and retry the refresh operation starting from the beginning.

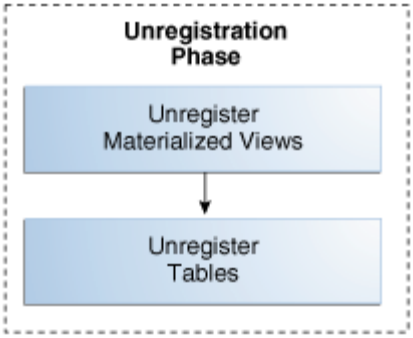
Figure 8–2 Refresh Phase



The Unregistration Phase

If you choose to stop using synchronous refresh, then you must unregister the materialized views as shown in Figure 8–3. The materialized views are first unregistered with the UNREGISTER_MVIEWS procedure. The tables are then unregistered by either dropping their staging logs or altering the staging logs to ordinary logs. Note that if the staging logs are converted to be ordinary materialized view logs with an ALTER MATERIALIZED LOG ... FOR FAST REFRESH statement, then the materialized views can be maintained with standard fast-refresh methods.

Figure 8–3 Unregistration Phase



Example 8–3 illustrates how to unregister the single materialized view MV1.

Example 8–3 Unregister Materialized Views

```
EXECUTE DBMS_SYNC_REFRESH.UNREGISTER_MVIEWS('MV1');
```

You can unregister multiple materialized views at one time:

```
EXECUTE DBMS_SYNC_REFRESH.UNREGISTER_MVIEWS('mv2, mv2_year, mv1_halfmonth');
```

You can verify to see that a materialized view has been unregistered by querying the DBA_SR_OBJ_ALL view.

Example 8–4 illustrates how to drop the staging log.

Example 8–4 Unregister Tables

```
DROP MATERIALIZED VIEW LOG ON fact;
```

Or you can alter the table to a materialized view log:

```
ALTER MATERIALIZED VIEW LOG ON fact
FOR FAST REFRESH;
```

You can verify to see that a table has been unregistered by querying the DBA_SR_OBJ_ALL view.

Using Synchronous Refresh Groups

The distinguishing feature of synchronous refresh is that changes to a table and its materialized views are loaded and refreshed together, hence the name synchronous refresh. For tables and materialized views to be maintained by synchronous refresh, the objects must be registered. Tables are registered for synchronous refresh when staging logs are created on them, and materialized views are registered using the REGISTER_MVIEWS procedure.

Synchronous refresh supports the refresh of materialized views built on multiple tables, with changes in one or more of them. Tables that are related by constraints must all necessarily be refreshed together to ensure data integrity. Furthermore, it is possible that some of the tables registered for synchronous refresh have several materialized views built on top of them, in which case, all those materialized views must also be refreshed together.

Instead of having you keep track of these dependencies, and issue the refresh commands on the right set of tables, Oracle Database automatically generates the

minimal sets of tables and materialized views that must necessarily be refreshed together. These sets are termed synchronous refresh groups or just sync refresh groups. Each sync refresh group is identified by a GROUP_ID.value.

The three procedures related to performing synchronous refresh (PREPARE_REFRESH, EXECUTE_REFRESH and ABORT_REFRESH) take as input either a single group ID or a list of group IDs identifying the sync refresh groups.

Each table or materialized view registered for synchronous refresh is assigned a GROUP_ID value, which may change over time, if the dependencies among them change. This happens when you issue the REGISTER_MVIEWS and UNREGISTER_MVIEWS procedures. The examples that follow show the sync refresh groups in a number of scenarios.

Because the GROUP_ID value can change with time, Oracle recommends the actual GROUP_ID value not be used when invoking the synchronous refresh procedures, but that the function DBMS_SYNC_REFRESH.GET_GROUP_ID be used instead. This function takes a materialized view name as input and returns the materialized view's GROUP_ID value.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about how to use the DBMS_SYNC_REFRESH.REGISTER_MVIEWS procedure

This section contains the following topics:

- Examples of Common Actions with Synchronous Refresh Groups
- Examples of Working with Multiple Synchronous Refresh Groups

Examples of Common Actions with Synchronous Refresh Groups

The synchronous refresh demo scripts in the rdbms/demo directory enable you to view typical operations that you are likely to perform. The main script is syncref_run.sql, and its log is syncref_run.log. Example 8–5, Example 8–6, and Example 8–7 below illustrate the different contexts in which the GET_GROUP_ID function can be used.

Example 8–5 illustrates how to display the objects registered in a group after registering them.

Example 8–5 Display the Objects Registered in a Group

```
EXECUTE DBMS_SYNC_REFRESH.REGISTER_MVIEWS('MV1');
SELECT NAME, TYPE, STAGING_LOG_NAME FROM USER_SR_OBJ
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME       TYPE       STAGING_LOG_NAME
---------- ---------- ----------------
MV1        MVIEW
FACT       TABLE      ST_FACT
STORE      TABLE      ST_STORE
TIME       TABLE      ST_TIME
```

Example 8–6 illustrates how to invoke refresh operations.

Example 8–6 Invoke Refresh Operations

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH( -
        DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
```

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH( -
        DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
SELECT NAME, TYPE, STATUS FROM USER_SR_OBJ_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;
```

Example 8–7 illustrates how to verify the status of objects registered in a group after an EXECUTE_REFRESH operation.

Example 8–7 Verify the Status of Objects Registered in a Group

```
SELECT NAME, TYPE, STATUS FROM USER_SR_OBJ_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME            TYPE       STATUS
--------------- ---------- ----------------
MV1             MVIEW      COMPLETE
FACT            TABLE      COMPLETE
STORE           TABLE      COMPLETE
TIME            TABLE      COMPLETE
```

Examples of Working with Multiple Synchronous Refresh Groups

You can work with multiple refresh groups at one time with the following APIs:

- GET_GROUP_ID_LIST

  Takes a list of materialized views as input and returns their group IDs in a list.

- GET_ALL_GROUP_IDS

  Returns the group IDs of all groups in the system in a list.

- The prepare refresh procedures (PREPARE_REFRESH, EXECUTE_REFRESH, and ABORT_REFRESH) can work multiple groups. Their overloaded versions accept lists of group IDs at a time.

Example 8–8 illustrates how to prepare the sync refresh groups of MV1, MV2, and MV3.

Example 8–8 Prepare Sync Refresh Groups

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(
        DBMS_SYNC_REFRESH.GET_GROUP_ID_LIST('MV1, MV2, MV3'));
```

Note that it is not necessary that these three materialized views be all in different groups. It is possible that two of the materialized views are in one group, and third in another; or even that all three materialized views are in the same group. Because PREPARE_REFRESH is overloaded to accept either a group ID or a list of group IDs, the above call will work in all cases.

Example 8–9 illustrates how to prepare and execute the refresh of all sync refresh groups in the system.

Example 8–9 Execute Sync Refresh Groups

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(
        DBMS_SYNC_REFRESH.GET_ALL_GROUP_IDS);

EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH(
        DBMS_SYNC_REFRESH.GET_ALL_GROUP_IDS);
```

Specifying and Preparing Change Data

Synchronous refresh requires you to specify and prepare the change data that serves as the input to the PREPARE_REFRESH and EXECUTE_REFRESH procedures. There are two methods for specifying the change data:

- Provide the change data in an outside table and register it with the REGISTER_PARTITION_OPERATION procedure.
- Provide the change data by in staging logs and process them with the PREPARE_STAGING_LOG procedure.

Some important points about change data are:

- The two methods are not mutually exclusive and can be employed at the same time, even on the same table, but there cannot be any conflicts in the changes specified. For instance, you can use the staging log to specify the change in a partition with a small number of changes, but if another partition has extensive changes, you can provide the changes for that partition in an outside table.
- For dimension tables, you can use only the staging logs to provide changes.
- Synchronous refresh can handle arbitrary combinations of changes in fact and dimension tables, but it is optimized for the most common data warehouse usage scenarios, where the bulk of the changes are made to only a few partitions of the fact table.
- Synchronous refresh places no restrictions on the use of nondestructive partition maintenance operations (PMOPS), such as add partition, used commonly in data warehouses. The use of such PMOPS is not directly related to the method used to specify change data.
- Synchronous refresh requires that all staging logs in the group must be prepared, even if the staging log has no changes registered in it.

This section contains the following topics:

- Working with Partition Operations
- Working with Staging Logs

Working with Partition Operations

Using the REGISTER_PARTITION_OPERATION procedure, you can provide the change data directly. This method is applicable only to fact tables. For each fact table partition that is changed, you must provide an outside table containing the data for that partition. The synchronous refresh demo (syncref_run.sql and syncref_run.log) contains an example. The steps are:

1. Create an outside table for the partition that it is intended to replace. It must have the same constraints as the fact table, and can be created in any desired tablespace.

```
CREATE TABLE      fact_ot_fp3(
   time_key       DATE NOT NULL REFERENCES time(time_key),
   store_key      INTEGER NOT NULL REFERENCES store(store_key),
   dollar_sales   NUMBER (6,2),
   unit_sales     INTEGER)
   tablespace     syncref_fp3_tbs;
```

2. Insert the data for this partition into the outside table.
3. Register this table for partition exchange.

```
begin
```

```
      DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION(
          partition_op                => 'EXCHANGE',
          schema_name                 => 'SYNCREF_USER',
          base_table_name             => 'FACT',
          partition_name              => 'FP3',
          outside_partn_table_schema  => 'SYNCREF_USER',
          outside_partn_table_name    => 'FACT_OT_FP3');
      end;
     /
    \
```

When you register the outside table and execute the refresh, Oracle Database performs the following operation at EXECUTE_REFRESH time:

```
ALTER TABLE FACT EXCHANGE PARTITION fp3 WITH TABLE fact_ot_fp3
INCLUDING INDEXES WITHOUT VALIDATION;
```

However, you are not allowed to issue the above statement directly on your own. If you do, Oracle Database will give this error:

```
ORA-31908: Cannot modify the contents of a table with a staging log.
```

Besides the EXCHANGE operation, the two other partition operations that can be registered with the REGISTER_PARTITION_OPERATION procedure are DROP and TRUNCATE.

Example 8–10 illustrates how to specify the drop of the first partition (FP1), by using the following statement.

Example 8–10 Registering a DROP Operation

```
begin
 DBMS_SYNC_REFRESH.REGISTER_PARTITION_OPERATION(
              partition_op                => 'DROP',
              schema_name                 => 'SYNCREF_USER',
              base_table_name             => 'FACT',
              partition_name              => 'FP1');
end;
/
```

If you wanted to truncate the partition instead, you could specify TRUNCATE instead of DROP for the partition_op parameter.

The three partition operations (EXCHANGE, DROP, and TRUNCATE) are called destructive PMOPS because they modify the contents of the table. The following partition operations are not destructive, and can be performed directly on a table registered with synchronous refresh:

- ADD PARTITION
- SPLIT PARTITION
- MERGE PARTITIONS
- MOVE PARTITION
- RENAME PARTITION

In data warehouses, these partition operations are commonly used to manage the large volumes of data, and synchronous refresh places no restrictions on their usage. Oracle Database requires only that these operations be performed before the PREPARE_REFRESH command is issued. This is because the PREPARE_REFRESH procedure computes the mapping between the fact table partitions and the materialized view partitions, and if any partition-maintenance is done between the PREPARE_REFRESH and EXECUTE_

REFRESH procedures, Oracle Database will detect this at EXECUTE_REFRESH and show an error.

You can use the USER_SR_PARTN_OPS catalog view to display the registered partition operations.

```
SELECT   TABLE_NAME, PARTITION_OP, PARTITION_NAME,
         OUTSIDE_TABLE_SCHEMA ot_schema, OUTSIDE_TABLE_NAME ot_name
FROM     USER_SR_PARTN_OPS
ORDER BY TABLE_NAME;

TABLE_NAME PARTITION_ PARTITION_NAME  OT_SCHEMA       OT_NAME
---------- ---------- --------------- --------------- --------------------
FACT       EXCHANGE   FP3             SYNCREF_USER    FACT_OT_FP3

1 row selected.
```

These partition operations are consumed by the synchronous refresh operation and are automatically unregistered by the EXECUTE_REFRESH procedure. So if you query USER_SR_PARTN_OPS after EXECUTE_REFRESH, it will show no rows.

After registering a partition, if you find you made a mistake or change your mind, you can undo it with the UNREGISTER_PARTITION_OPERATION command:

```
begin
 DBMS_SYNC_REFRESH.UNREGISTER_PARTITION_OPERATION(
              partition_op                  => 'EXCHANGE',
              schema_name                   => 'SYNCREF_USER',
              base_table_name               => 'FACT',
              partition_name                => 'FP3');
end;
/
```

Working with Staging Logs

In synchronous refresh, staging logs play a role similar to materialized view logs in incremental refresh. They are created with a DDL statement and can be altered to a materialized view log. Unlike materialized view logs, however, you are responsible for loading changes into the staging logs in a specified format. Each row in the staging log must have a key to identify it uniquely; this key is called the **staging log key**, and is defined in "Staging Log Key" on page 8-15.

You are responsible for populating the staging log, which will consist of all the columns in the base table and an additional control column DMLTYPE$$ of type CHAR(2). This must have the value 'I' to denote the row is being inserted, 'D' for delete, and 'UN' and 'UO' for the new and old values of the row being updated, respectively. The last two must occur in pairs.

The staging log is validated by the PREPARE_STAGING_LOG procedure and consumed by the synchronous refresh operations (PREPARE_REFRESH and EXECUTE_REFRESH). During validation by PREPARE_STAGING_LOG, if errors are detected, they will be captured in an exceptions table. You can query the view USER_SR_STLOG_EXCEPTIONS to get details on the exceptions.

Synchronous refresh requires that, before calling PREPARE_REFRESH for sync refresh groups, the staging logs of all tables in the group must be processed with PREPARE_STAGING_LOG. This is necessary even if a table has no change data and its staging log is empty.

This section contains the following topics:

Staging Log Key

In order to create a staging log on a base table, the base table must have a key. If the table has a primary key, the primary key is deemed to be staging log key on the table's staging log. Note that every dimension table has a primary key.

With fact tables, it is less common for them to have a primary key. If a table does not have a primary key, the columns that are the foreign keys of its dimension tables constitute its staging log key.

The key of a staging log can be described as:

- The primary key of the base table. If a fact table has a primary key, it is sometimes called a surrogate key.
- The set of foreign keys for a fact table. This applies if the fact table does not have a primary key. This assumption is common in data warehouses, though it is not enforced.

The rules for loading staging logs are described in "Staging Log Rules" on page 8-15.

The `PREPARE_STAGING_LOG` procedure verifies that each key value is specified at most once. When populating the staging log, it is your responsibility to consolidate the changes if a row with the same key value is changed more than once. This process is known as **change consolidation**. When doing the change consolidation, you must:

- Consolidate a delete-insert of the same row into an update operation with rows `'UO'` and `'UN'`.
- Consolidate multiple updates into a single update.
- Prevent null changes such as an insert-update-delete of the same row from appearing in the staging log.
- Consolidate an insert followed by multiple updates into a single insert.

Staging Log Rules

Every row should contain non-null values for all the columns comprising the primary key. You are required to consolidate all the changes so that each key in the staging log can be specified only for one type of operation.

For the rows being inserted (`DMLTYPE$$` is `'I'`), all columns in the staging log must be supplied with valid values, conforming to any constraint on the corresponding columns in the base table. Keys of rows being inserted must not exist in the base table.

For the rows being deleted (`DMLTYPE$$` is `'D'`), the non-key column values are optional. Similarly, for the rows specifying the old values of the columns being updated (`DMLTYPE$$` is `'UO'`), the non-key column values are optional; an important exception is the column whose values are being updated to `NULL`, as explained subsequently.

For the rows specifying the new values of the columns being updated (`DMLTYPE$$` is `'UN'`), the non-key column values are optional except for the values of the columns that were changed.

Columns Being Updated to NULL

If a column is being updated to NULL, its old value must be specified. Otherwise, Oracle Database may not be able to distinguish this from a column whose value is being left unchanged in the update.

For example, let table T1 have three columns c1, c2, and c3. Let there be a row with (c1, c2, c3) = (1, 5, 10), and you supply the following information in the staging log:

DMLTYPE$$	C1	C2	C3
UO	1	NULL	NULL
UN	1	NULL	11

The result would be that the new row could be (1, 5, 11) or (1, NULL, 11) without having specified the old value. However, with that specification, it is clear the new row is (1, 5, 11). If you want to specify NULL for c2, you should specify the old value in the UO row as follows:

DMLTYPE$$	C1	C2	C3
UO	1	5	NULL
UN	1	NULL	11

Because the old value of c2 is 5, (the correct previously updated value for the column), its new value, will be NULL and the new row is (1, NULL, 11).

Examples of Working with Staging Logs

This section illustrates examples of working with staging logs.

The PREPARE_STAGING_LOG procedure has an optional third parameter called PSL_MODE. This allows you to specify whether any or all of the three types of DML statements specified in the staging log can be treated as trusted, and not be subject to verification by the PREPARE_STAGING_LOG procedure, as shown in Example 8–11.

Example 8–11 Specifying Trusted DML Statements

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store',
                                 DBMS_SYNC_REFRESH.INSERT_TRUSTED +
                                 DBMS_SYNC_REFRESH.DELETE_TRUSTED);
```

This call will skip verification of INSERT and DELETE DML statements in the staging log of STORE but will verify UPDATE DML statements.

Example 8–12 is taken from the demo syncref_run.sql. It shows that the user has provided values for all columns for the delete and update operations. This is recommended if these values are available.

Example 8–12 Preparing Staging Logs

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('I',   5, 5, 'Store 5', '03060');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('I',   6, 6, 'Store 6', '03062');
```

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UO',  4, 4, 'Store 4', '03062');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UN',  4, 4, 'Stor4NewNam', '03062');

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('D',   3, 3, 'Store 3', '03060');

EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

-- display initial contents of st_store

SELECT dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE
FROM st_store
ORDER BY STORE_KEY ASC, dmltype$$ DESC;

DM  STORE_KEY  STORE_NUMBER  STORE_NAME    ZIPCODE
--  ---------  ------------  ----------    -------
D            3             3  Store 3         03060
UO           4             4  Store 4         03062
UN           4             4  Stor4NewNam     03062
I            5             5  Store 5         03060
I            5             5  Store 6         03062

5 rows selected.
```

Example 8–13 shows that if you do not supply all the values for the delete and update operations, then when you run the PREPARE_STAGING_LOG procedure, Oracle Database will fill in missing values.

Example 8–13 Filling in Missing Values for Deleting and Updating Records

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('D',  3, NULL, NULL, NULL);

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UO',  4, NULL, NULL, NULL);

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UN',  4, NULL, NULL, '03063');

EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

SELECT dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE
FROM ST_STORE ORDER BY STORE_KEY ASC, dmltype$$ DESC;

DM STORE_KEY STORE_NUMBER STORE_NAME  ZIPCODE
-- --------- ------------ ----------- ---------
D          3            3 Store 3     03060
UO         4            4 Store 4     03062
UN         4            4 Store 4     03063
```

Example 8–14 illustrates how to update a column to NULL. If you want to update a column value to NULL, then you must provide its old value in the UO record.

Example 8–14 Updating a Column to NULL

In this example, your goal is to change the zipcode of store 4 to 03063 and its name to NULL. You can supply the old zipcode value, but you must supply the old value of store_name in the 'UO' row, or else store_name will be unchanged.

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UO',  4, NULL, 'Store 4', NULL);

INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('UN',  4, NULL, NULL, '03063');

EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

SELECT dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE
FROM st_store ORDER BY STORE_KEY ASC, dmltype$$ DESC;

DM STORE_KEY STORE_NUMBER STORE_NAME  ZIPCODE
-- --------- ------------ ----------- --------
UO         4            4 Store 4     03062
UN         4            4             03063
```

Example 8–15 illustrates how to use the USER_SR_STLOG_STATS catalog view to display the staging log statistics.

Example 8–15 Displaying Staging Log Statistics

```
SELECT   TABLE_NAME, STAGING_LOG_NAME, NUM_INSERTS, NUM_DELETE, NUM_UPDATES
FROM     USER_SR_STLOG_STATS
ORDER BY TABLE_NAME;

TABLE_NAME STAGING_LOG_NAME NUM_INSERTS NUM_DELETES NUM_UPDATES
---------- ---------------- ----------- ----------- -----------
FACT       ST_FACT                    4           1           1
STORE      ST_STORE                   2           1           1
TIME       ST_TIME                    1           0           0

3 rows selected.
```

If you use the same query at the end of the EXECUTE_REFRESH procedure, then you will get no rows, indicating the change data has all been consumed by synchronous refresh.

Error Handling in Preparing Staging Logs

When a table is processed by the PREPARE_STAGING_LOG procedure, it will detect and report errors in the specification of change data that relates only to that table. For example, it will verify that keys of rows being inserted do not already exist in the base table and that keys of rows being deleted or updated do exist. However, the PREPARE_STAGING_LOG procedure cannot detect errors related to the referential integrity constraints on the table; that is, it cannot detect errors if there are inconsistencies in the specification of change data that involves more than one table. Such errors will be detected at the time of the EXECUTE_REFRESH procedure.

Troubleshooting Synchronous Refresh Operations

This section describes how to monitor the status of the two synchronous refresh procedures, PREPARE_REFRESH and EXECUTE_REFRESH and how to troubleshoot errors

that may occur. To be successful in using synchronous refresh, you should be aware of the different types of errors that can arise and how to deal with them.

One of the most likely sources of errors is from incorrect preparation of the change data. These errors will present themselves as referential constraint violations when the EXECUTE_REFRESH procedure is run. In such cases, the status of the group is set to ABORT. It is important to learn to recognize these errors and address them.

The topics covered in this section are:

Overview of the Status of Refresh Operations

The DBMS_SYNC_REFRESH package provides three procedures to control the refresh execution process. You initiate synchronous refresh with the PREPARE_REFRESH procedure, which plans the entire refresh operation and does the bulk of the computational work for refresh, followed by the EXECUTE_REFRESH procedure, which carries out the refresh. The third procedure provided is ABORT_REFRESH, which is used to recover from errors if either of these procedures fails.

The USER_SR_GRP_STATUS and USER_SR_OBJ_STATUS catalog views contain all the information on the status of these refresh operations for current groups:

- The USER_SR_GRP_STATUS view shows the status of the group as a whole.
  - The OPERATION field indicates the current refresh procedure run on the group: PREPARE or EXECUTE.
  - The STATUS field indicates the status of the operation - RUNNING, COMPLETE, ERROR-SOFT, ERROR-HARD, ABORT, PARTIAL. These are explained in detail later.
  - The group is identified by its group ID.
- The USER_SR_OBJ_STATUS view shows the status of each individual object.
  - The object is identified by its owner, name, and type (TABLE or MVIEW) and group ID.
  - The STATUS field, which may be NOT PROCESSED, ABORT, or COMPLETE. These are explained in detail later.

How PREPARE_REFRESH Sets the STATUS Fields

When you launch a new PREPARE_REFRESH job, the group's STATUS is set to RUNNING and the STATUS of the objects in the group is set to NOT PROCESSED. When the PREPARE_REFRESH job finishes, the status of the objects remains unchanged, but the group's status is changed to one of following three values:

- COMPLETE if the job completed successfully.
- ERROR_SOFT if the job encountered the ORA-01536: space quota exceeded for tablespace '%s' error.

- ERROR_HARD otherwise (that is, if the job encountered any error other than ORA-01536).

Some points to keep in mind when using the PREPARE_REFRESH procedure:

- The NOT PROCESSED status of the objects in the group signifies that the data of the objects has not been modified by the PREPARE_REFRESH job. The data modification will occur only in the EXECUTE_REFRESH step, at which time the status will be changed as appropriate. This is described later.
- If the STATUS is ERROR_SOFT, you can fix the ORA-01536 error by increasing the space quota for the specified tablespace, and resume PREPARE_REFRESH. Alternatively, you can choose to abort the refresh with ABORT_REFRESH.
- If the STATUS value is ERROR_HARD, then your only option is to abort the refresh with ABORT_REFRESH.
- If the STATUS value after the PREPARE_REFRESH procedure finishes is RUNNING, then an error has occurred. Contact Oracle Support Services for assistance.
- A STATUS value of ERROR_HARD might be related to running out of resources because the PREPARE_REFRESH procedure can be resource-intensive. If you are not able to identify the problem, then contact Oracle Support Services for assistance. But if you can identify the problem and fix it, then you might be able to continue using synchronous refresh, by first running ABORT_REFRESH and then the PREPARE_REFRESH procedure.
- Remember that you can launch a new PREPARE_REFRESH job only when the previous refresh operation on the group (if any) has either completed execution successfully or has aborted.
- If the STATUS value of the PREPARE_REFRESH procedure at the end is not COMPLETE, you cannot proceed to the EXECUTE_REFRESH step. If you are unable to get PREPARE_REFRESH to work correctly, then you can proceed to the unregistration phase, and maintain the objects in the groups with other refresh methods.

Examples of PREPARE_REFRESH

This section offers examples of common cases when preparing a refresh.

Example 8–16 shows a PREPARE_REFRESH procedure completing successfully.

Example 8–16 PREPARE_REFRESH Succeeds with Status COMPLETE

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    COMPLETE
```

Example 8–17 shows a PREPARE_REFRESH procedure encountering ORA-01536.

Example 8–17 PREPARE_REFRESH Fails with Status ERROR_SOFT

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
BEGIN DBMS_SYNC_REFRESH.PREPARE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
```

```
END;

*
ERROR at line 1:
ORA-01536: space quota exceeded for tablespace 'DUMMY_TS'
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 63
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 411
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 429
ORA-06512: at line 1PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    ERROR_SOFT
```

Example 8–18 is a continuation of Example 8–2. After the ORA-01536 error is raised, increase the tablespace for DUMMY_TS and rerun the PREPARE_REFRESH procedure, which now completes successfully. Note that the PREPARE_REFRESH procedure will resume processing from the place where it stopped. Also note the usage of the PREPARE_REFRESH procedure is no different from normal, and does not require any parameters or settings to indicate the procedure is being resumed.

Example 8–18 Resume of PREPARE_REFRESH Succeeds

```
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    COMPLETE
```

Example 8–19 assumes the PREPARE_REFRESH procedure has failed and the STATUS value is ERROR_HARD. You then run the ABORT_REFRESH procedure to abort the prepare job. Note that the STATUS value has changed from ERROR_HARD to ABORT at the end.

Example 8–19 Abort of PREPARE_REFRESH

```
SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
PREPARE    ERROR_HARD

EXECUTE DBMS_SYNC_REFRESH.ABORT_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');
```

```
OPERATION  STATUS
---------  ------
PREPARE    ABORT
```

How EXECUTE_REFRESH Sets the Status Fields

The EXECUTE_REFRESH procedure divides the group of objects in the sync refresh group into subgroups, each of which is refreshed atomically. The first subgroup consists of the base tables. Each materialized view in the sync refresh group is placed in a separate subgroup and refreshed atomically.

In the case of the EXECUTE_REFRESH procedure, the possible end states of the STATUS field are: COMPLETE, PARTIAL, and ABORT:

- STATUS = COMPLETE

  This state is reached if the base tables and all the materialized views refresh successfully.

- STATUS = ABORT

  This state indicates the refresh of the base tables subgroup has failed; the data in the tables and materialized views is consistent but unchanged. If this happens, then there should be an error associated with the failure. If it is a user error, such as a constraint violation, then you can fix the problem and retry the synchronous refresh operation from the beginning (that is, PREPARE_STAGING_LOG for each table in the group PREPARE_REFRESH and EXECUTE_REFRESH.). If it is not a user error, then you should contact Oracle Support Services.

- STATUS = PARTIAL

  If all the base tables refresh successfully and some, but not all, materialized views refresh successfully, then this state is reached. The data in the tables and materialized views that have refreshed successfully are consistent with one another; the other materialized views are stale and need complete refresh. If this happens, there should be an error associated with the failure. Most likely this is not a user error, but an Oracle error that you should report to Oracle Support Services. You have two choices in this state:

  – Retry execution of the EXECUTE_REFRESH procedure. In such a case, EXECUTE_REFRESH will retry the refresh of the failed materialized views with another refresh method like PCT-refresh or COMPLETE refresh. If all materialized views succeed, then the status will be set to COMPLETE. Otherwise, the status will remain at PARTIAL.

  – Invoke the ABORT_REFRESH procedure to abort the materialized views. This will roll back changes to all materialized views and base tables. They will all have the same data as in the original state before any of the changes in the staging logs or registered partition operations has been applied to them.

In the case of errors in the EXECUTE_REFRESH procedure, the following fields in the USER_SR_GRP_STATUS view are also useful:

- NUM_MVS_COMPLETED, which contains the number of materialized views that completed the refresh operation successfully.
- NUM_MVS_ABORTED, which contains the number of materialized views that aborted.
- ERROR and ERROR_MESSAGE, which records the error encountered in the operation.

At the end of the EXECUTE_REFRESH, procedure, the statuses of the objects in the group are marked as follows in the USER_SR_OBJ_STATUS view:

- The status of an object is set to COMPLETE if the changes were applied to it successfully.
- The status of an object is set to ABORT if the changes were not applied successfully. In this case, the object will be in the same state as it was before the refresh operation. The ERROR and ERROR_MESSAGE fields record the error encountered in the operation.
- The status of an object remains NOT PROCESSED if no changes were applied to it.

Examples of EXECUTE_REFRESH

This section provides examples of common cases when executing a refresh.

Example 8–20 shows an EXECUTE_REFRESH procedure completing successfully.

Example 8–20 EXECUTE_REFRESH Completes Successfully

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH( DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ------
EXECUTE    COMPLETE
```

Example 8–21 shows an EXECUTE_REFRESH procedure succeeding partially. In this example, the EXECUTE_REFRESH procedure fails after refreshing the base tables but before completing the refresh of all the materialized views. The resulting status of the group is PARTIAL and the QSM-03280 error message is thrown.

Example 8–21 EXECUTE_REFRESH Succeeds Partially

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
BEGIN DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
END;

*
ERROR at line 1:
ORA-31928: Synchronous refresh error
QSM-03280: One or more materialized views failed to refresh successfully.
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 63
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 411
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 446
ORA-06512: at line 1
```

Check the status of the group itself after the EXECUTE_REFRESH.procedure. Note that the operation field is set to EXECUTE and the status is PARTIAL.

```
SELECT OPERATION, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  -------------
EXECUTE    PARTIAL
```

By querying the USER_SR_GRP_STATUS view, you find the number of materialized views that have aborted is 1 and the failed materialized view is MV1.

If you examine the status of objects in the group, because STORE and TIME are unchanged, then their status is NOT PROCESSED.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_OBJ_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME            TYPE       STATUS
--------------- ---------- ----------------
MV1             MVIEW      ABORT
MV1_HALFMONTH   MVIEW      COMPLETE
MV2             MVIEW      COMPLETE
MV2_YEAR        MVIEW      COMPLETE
FACT            TABLE      COMPLETE
STORE           TABLE      NOT PROCESSED
TIME            TABLE      NOT PROCESSED

7 rows selected.

SELECT NUM_TBLS, NUM_MVS, NUM_MVS_COMPLETED, NUM_MVS_ABORTED
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

NUM_TBLS NUM_MVS NUM_MVS_COMPLETED NUM_MVS_ABORTED
-------- ------- ----------------- ---------------
       3       4                 3               1
```

At this point, you can attempt to run the EXECUTE_REFRESH procedure once more. If the retry succeeds and the failed materialized views succeed, then the group status will be set to COMPLETE. Otherwise, the status will remain at PARTIAL. This is shown in Example 8–22. You can also abort the refresh procedure and return to the original state. This is shown in Example 8–23.

Example 8–22 illustrates a continuation of Example 8–21. You retry the EXECUTE_REFRESH procedure and it succeeds:

Example 8–22 Retrying a Refresh After a PARTIAL Status

```
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.

--Check the status of the group itself after the EXECUTE_REFRESH operation;
--note that the operation field is set to EXECUTE and status is COMPLETE.

SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------  ---------
EXECUTE    COMPLETE
```

By querying the USER_SR_GRP_STATUS view, you find the number of materialized views that have aborted is 0 and the status of MV1 is COMPLETE. If you examine the status of objects in the group, because STORE and TIME are unchanged, then their status is NOT PROCESSED.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME             TYPE       STATUS
---------------- ---------- ----------------
MV1              MVIEW      COMPLETE
MV1_HALFMONTH    MVIEW      COMPLETE
MV2              MVIEW      COMPLETE
MV2_YEAR         MVIEW      COMPLETE
FACT             TABLE      COMPLETE
STORE            TABLE      NOT PROCESSED
TIME             TABLE      NOT PROCESSED

7 rows selected.

SELECT NUM_TBLS, NUM_MVS, NUM_MVS_COMPLETED, NUM_MVS_ABORTED
FROM  USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

NUM_TBLS NUM_MVS NUM_MVS_COMPLETED NUM_MVS_ABORTED
-------- ------- ----------------- ---------------
       3       4                 4               0
```

You can examine the tables and materialized views to verify that the changes in the change data have been applied to them correctly, and the materialized views and tables are consistent with one another.

Example 8–23 illustrates aborting a refresh procedure that is in a PARTIAL state.

Example 8–23 Aborting a Refresh with a PARTIAL Status

```
EXECUTE DBMS_SYNC_REFRESH.ABORT_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

PL/SQL procedure successfully completed.
```

Check the status of the group itself after the ABORT_REFRESH procedure; note that the operation field is set to EXECUTE and status is ABORT.

```
SELECT OPERATION, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------- -------
EXECUTE    ABORT
```

By querying the USER_SR_GRP_STATUS view, you see that all the materialized views have aborted, and the fact table as well. Check the status of objects in the group; because STORE and TIME are unchanged, their status is NOT PROCESSED.

```
SELECT NAME, TYPE, STATUS FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1')
ORDER BY TYPE, NAME;

NAME             TYPE       STATUS
---------------- ---------- ----------------
MV1              MVIEW      ABORT
MV1_HALFMONTH    MVIEW      ABORT
MV2              MVIEW      ABORT
MV2_YEAR         MVIEW      ABORT
FACT             TABLE      ABORT
```

```
STORE            TABLE      NOT PROCESSED
TIME             TABLE      NOT PROCESSED

7 rows selected.

SELECT NUM_TBLS, NUM_MVS, NUM_MVS_COMPLETED, NUM_MVS_ABORTED
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

NUM_TBLS NUM_MVS NUM_MVS_COMPLETED NUM_MVS_ABORTED
-------- ------- ----------------- ---------------
       3       4                 0               4
```

You can examine the tables and materialized views to verify that they are all in the original state and no changes from the change data have been applied to them.

Example of EXECUTE_REFRESH with Constraint Violations

In the synchronous refresh method, change data is loaded into the tables and materialized views at the same time to keep them synchronized. In the other refresh methods, change data is loaded into tables first, and any constraints that are enabled are checked at that time. In the synchronous refresh method, the outside table is prepared using trusted data from the user, and constraint validation is turned off to save execution time. The following example shows a constraint violation that is caught by the EXECUTE_REFRESH procedure. In such cases, the final status of the EXECUTE_REFRESH procedure will be ABORT. You will have to identify and fix the problem in the change data and begin the synchronous refresh phase all over.

Example 8–24 Child Key Constraint Violation

In Example 8–24, assume the same tables as in the file syncref_run.sql in the rdbms/demo directory are used and populated with the same data. In particular, the table STORE has four rows with the primary key STORE_KEY having the values 1 through 4, and the FACT table has rows referencing all four stores, including store 3.

To demonstrate a parent-key constraint violation, populate the staging log of STORE with the delete of the row having the STORE_KEY of 3. There are no other changes to the other tables. When the EXECUTE_REFRESH procedure runs, it fails with the ORA-02292 error as shown.

```
INSERT INTO st_store (dmltype$$, STORE_KEY, STORE_NUMBER, STORE_NAME, ZIPCODE)
VALUES ('D',   3, 3, 'Store 3', '03060');

-- Prepare the staging logs
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'fact');
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'time');
EXECUTE DBMS_SYNC_REFRESH.PREPARE_STAGING_LOG('syncref_user', 'store');

-- Prepare the refresh
EXECUTE DBMS_SYNC_REFRESH.PREPARE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));

-- Execute the refresh
EXECUTE DBMS_SYNC_REFRESH.EXECUTE_REFRESH( -
          DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
BEGIN DBMS_SYNC_REFRESH.EXECUTE_REFRESH(DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1'));
END;

*
ERROR at line 1:
```

```
ORA-02292: integrity constraint (SYNCREF_USER.SYS_C0031765) violated - child
record found
ORA-06512: at line 1
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 63
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 411
ORA-06512: at "SYS.DBMS_SYNC_REFRESH", line 446
ORA-06512: at line 1
```

Examine the status of the group itself after the EXECUTE_REFRESH procedure. Note that the operation field is set to EXECUTE and the status is ABORT.

```
SELECT OPERATION, STATUS
FROM USER_SR_GRP_STATUS
WHERE GROUP_ID = DBMS_SYNC_REFRESH.GET_GROUP_ID('MV1');

OPERATION  STATUS
---------- --------------
EXECUTE    ABORT
```

If you check the contents of the base tables and of MV1, then you will find there is no change, and they all have the original values.

Performing Synchronous Refresh Eligibility Analysis

The CAN_SYNCREF_TABLE function tells you whether a table and its dependent materialized views are eligible for synchronous refresh. It provides an explanation of its analysis. If the table and views are not eligible, you can examine the reasons and take appropriate action if possible. To be eligible for synchronous refresh, a table must satisfy the various criteria described earlier.

You can invoke CAN_SYNCREF_TABLE function 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:

```
can_syncref_table(schema_name   IN VARCHAR2,
                  table_name    IN VARCHAR2,
                  statement_id  IN VARCHAR2)
```

You can create an output table called SYNCREF_TABLE by executing the utlcsrt.sql script. If you want to direct the output of the CAN_SYNCREF_TABLE function to a VARRAY instead of a table, then you should call the procedure as follows:

```
can_syncref_table(schema_name   IN VARCHAR2,
                  table_name    IN VARCHAR2,
                  output_array  IN OUT Sys.CanSyncRefTypeArray)
```

Table 8–1 CAN_SYNCREF_TABLE

Parameter	Description
schema_name	Name of the schema of the base table.
base_table_name	Name of the base table.
statement_id	A string (VARCHAR2(30) to identify the rows pertaining to a call of the CAN_SYNCREF_TABLE function when the output is directed to a table named SYNCREF_TABLE in the user's schema.
output_array	The output array into which CAN_SYNCREF_TABLE records the information on the eligibility of the base table and its dependent materialized views for synchronous refresh.

Note: Only one `statement_id` or `output_array` parameter need be provided to the `CAN_SYNCREF_TABLE` function.

This section contains the following topics:

- Using SYNCREF_TABLE
- Using a VARRAY
- Demo Scripts

Using SYNCREF_TABLE

The output of the `CAN_SYNCREF_TABLE` function can be directed to a table named `SYNCREF_TABLE`. You are responsible for creating `SYNCREF_TABLE`; it can be dropped when it is no longer needed. The format of `SYNCREF_TABLE` is as follows:

```
CREATE TABLE SYNCREF_TABLE(
          statement_id             VARCHAR2(30),
          schema_name              VARCHAR2(30),
          table_name               VARCHAR2(30),
          mv_schema_name           VARCHAR2(30),
          mv_name                  VARCHAR2(30),
          eligible                 VARCHAR2(1),   -- 'Y' , 'N'
          seq_num                  NUMBER,
          msg_number               NUMBER,
          message                  VARCHAR2(4000)
);
```

You must provide a different `statement_id` parameter for each invocation of this procedure on the same table. If not, an error will be thrown. The `statement_id`, `schema_name`, and `table_name` fields identify the results for a given table and `statement_id`.

Each row contains information on the eligibility of either the table or its dependent materialized view. The `CAN_SYNCREF_TABLE` function guarantees that each row has values for both `mv_schema_name` and `mv_name` that are either `NULL` or non-`NULL`. These rows have the following semantics:

- If the `mv_schema_name` value is `NULL` and `mv_name` is `NULL`, then the `ELIGIBLE` field describes whether the table is eligible for synchronous refresh; if the table is not eligible, the `MSG_NUMBER` and `MESSAGE` fields provide the reason for this.
- If the `mv_schema_name` value is `NOT NULL` and `mv_name` is `NOT NULL`, then the `ELIGIBLE` field describes whether the materialized view is eligible for synchronous refresh; if the materialized view is not eligible, the `MSG_NUMBER` and `MESSAGE` fields provide the reason for this.

You must provide a different `statement_id` parameter for each invocation of this procedure on the same table, or else an error will be thrown. The `statement_id`, `schema_name`, and `table_name` fields identify the results for a given table and `statement_id`.

Using a VARRAY

You can save the output of the `CAN_SYNCREF_TABLE` function in a PL/SQL `VARRAY`. The elements of this array are of type `CanSyncRefMessage`, which is predefined in the `SYS` schema as shown in the following example:

```
TYPE CanSyncRefMessage IS OBJECT (
          schema_name         VARCHAR2(30),
          table_name          VARCHAR2(30),
          mv_schema_name      VARCHAR2(30),
          mv_name             VARCHAR2(30),
          eligible            VARCHAR2(1),    -- 'Y' , 'N'
          seq_num             NUMBER,
          msg_number          NUMBER,
          message             VARCHAR2(4000)
);
```

The array type, `CanSyncRefArrayType`, which is a `VARRAY` of `CanSyncRefMessage` objects, is predefined in the `SYS` schema as follows:

```
TYPE  CanSyncRefArrayType AS VARRAY(256) OF CanSyncRefMessage;
```

Each `CanSyncRefMessage` record provides a message concerning the eligibility of the base table or a dependent materialized view for synchronous refresh. The semantics of the fields is the same as that of the corresponding fields in `SYNCREF_TABLE`. However, `SYNCREF_TABLE` has a `statement_id` field that is absent in `CanSyncRefMessage` because no `statement_id` is supplied (because it is not required) when the `CAN_SYNCREF_TABLE` procedure is called with a `VARRAY` parameter.

The default size limit for `CanSyncRefArrayType` is 256 elements. If you need more than 256 elements, then connect as `SYS` and redefine `CanSyncRefArray`. The following commands, when connected as the `SYS` user, redefine `CanSyncRefArray` and set the limit to 2048 elements:

```
CREATE OR REPLACE TYPE CanSyncRefArrayType AS VARRAY(2048) OF
SYS.CanSyncRefMessage;
/
GRANT EXECUTE ON SYS.CanSyncRefMessage TO PUBLIC;

CREATE OR REPLACE PUBLIC SYNONYM CanSyncRefMessage FOR SYS.CanSyncRefMessage;
/
GRANT EXECUTE ON SYS.CanSyncRefArrayType TO PUBLIC;

CREATE OR REPLACE PUBLIC SYNONYM CanSyncRefArrayType FOR SYS.CanSyncRefArrayType;
/
```

Demo Scripts

The synchronous refresh demo scripts in the `rdbms/demo` directory contain examples of the most common scenarios of the various synchronous refresh operations, including `CAN_SYNCREF_API`. The main script is `syncref_run.sql` and its log is `syncref_run.log`. The file `syncref_cst.sql` defines two procedures `DO_CST` and `DO_CST_ARR`, which simplify the usage of the `CAN_SYNCREF_TABLE` function and display the information on the screen in a convenient format. This format is documented in the `syncref_cst.sql` file.

Overview of Synchronous Refresh Security Considerations

The execute privilege on the `DBMS_SYNC_REFRESH` package is granted to `PUBLIC`, so all users can execute the procedures in that package to perform synchronous refresh on objects owned by them. The database administrator can perform synchronous refresh operation on all tables and materialized views in the database.

In general, if a user without the DBA privilege wants to use synchronous refresh on another user's table, he must have complete privileges to read from and write to that table; that is, the user must have the SELECT, INSERT, UPDATE, and DELETE privileges on that table or materialized view. The user can have the READ privilege instead of the SELECT privilege. A couple of exceptions occur in the following:

- PURGE_REFRESH_STATS and ALTER_REFRESH_STATS_RETENTION functions

  These two functions implement the purge policy and can be used to change the default retention period. These functions can be executed only by the database administrator.

- The CAN_SYNCREF_TABLE function

  This is an advisory function that examines the eligibility for synchronous refresh of all the materialized views associated with a specified table. Hence, this function requires the READ or SELECT privilege on all materialized views associated with the specified table.

9

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?

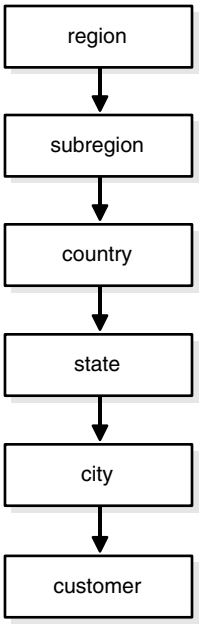
The data in the retailer's data warehouse system has two important components: dimensions and facts. The dimensions are products, customers, promotions, channels, and time. One approach for identifying your dimensions is to review your reference tables, such as a product table that contains everything 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).

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 9–1.

Figure 9–1 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.

Dimension schema objects (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.

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.

See Also:

- Chapter 4, "Data Warehousing Optimizations and Techniques" for further details about schemas
- Chapter 10, "Basic Query Rewrite for Materialized Views" for further details regarding query rewrite
- *Oracle Database SQL Tuning Guide* for further details regarding the SQL Access Advisor

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 9–1. 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.

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 9–1 on page 9-2.

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 9-8 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.

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 9-10.

See Also:

- Chapter 10, "Basic Query Rewrite for Materialized Views" for further details of using dimensional information
- *Oracle Database SQL Language Reference* for a complete description of the `CREATE DIMENSION` statement

This section contains the following topics:

- Dropping and Creating Attributes with Columns
- Multiple Hierarchies
- Using Normalized Dimension Tables

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);
```

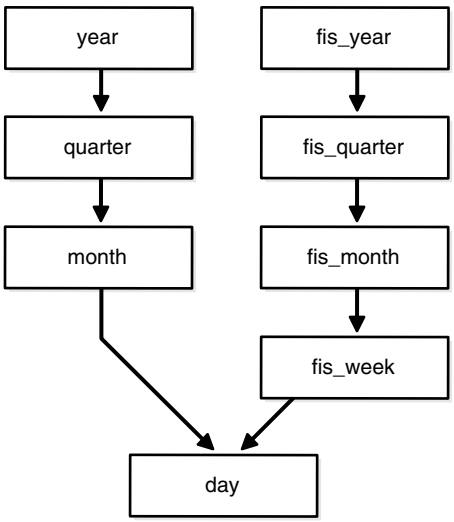
```
ALTER DIMENSION products_dim
DROP ATTRIBUTE subcategory_att LEVEL subcategory COLUMN prod_subcategory;
```

> **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 9–2 illustrates a dimension `times_dim` with two time hierarchies.

Figure 9–2 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:

- Viewing Dimensions With Oracle Enterprise Manager
- Viewing Dimensions With the DESCRIBE_DIMENSION Procedure

Viewing Dimensions With 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.

Viewing Dimensions With 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 10, "Basic Query Rewrite for Materialized Views" 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 9–2 on page 9-7, 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 9-6, 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;
```

10

Basic Query Rewrite for Materialized Views

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. "When Does Oracle Rewrite a Query?" on page 10-2 describes the conditions that must be met for a query to be rewritten.

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.

This section contains the following topics:

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 10-4).

  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 11-60 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.

11

Advanced Query Rewrite for Materialized Views

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.

You can use the following types of query rewrite: Text Match Rewrite or General Query Rewrite Methods.

This section discusses the optimizer in more detail and contains the following sections:

- Cost-Based Optimization
- General Query Rewrite Methods

- Checks Made by Query Rewrite
- Query Rewrite Using Dimensions

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.

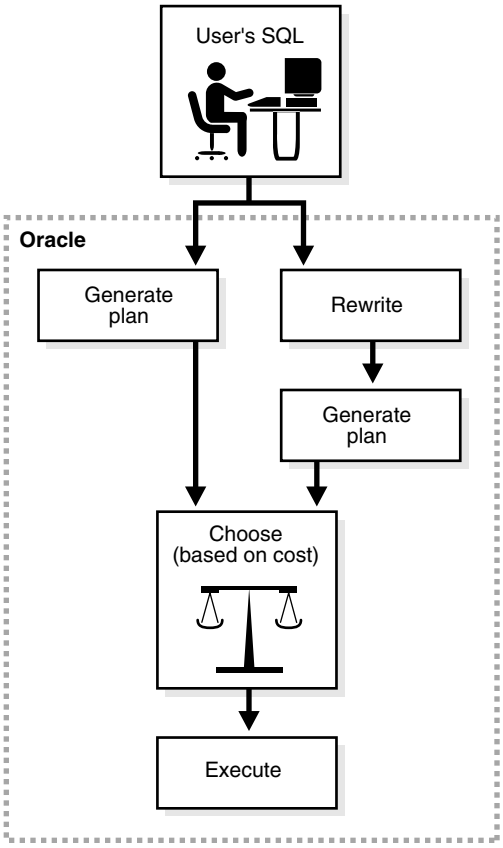
If query rewrite has a choice between several materialized views to rewrite a query block, it selects the ones which can result in reading in the least amount of data. After a materialized view has been selected for a rewrite, the optimizer then tests whether the rewritten query can be rewritten further 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 11–1 presents a graphical view of the cost-based approach used during the rewrite process.

Figure 11–1 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.

See Also: When are Constraints and Dimensions Needed?

When are Constraints and Dimensions Needed?

Table 11–1 illustrates when dimensions and constraints are required for different types of query rewrite. These types of query rewrite are described throughout this chapter.

Table 11–1 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

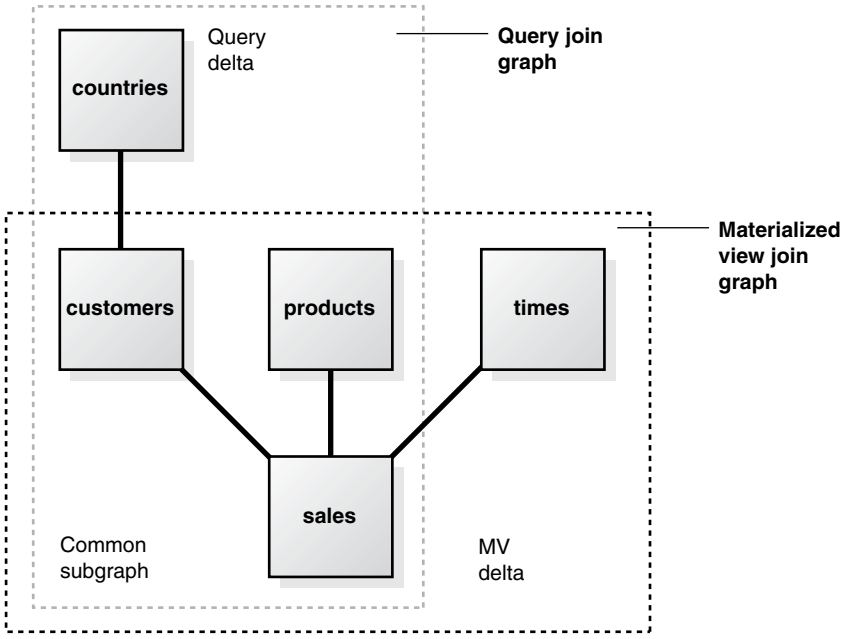
Join Compatibility Check

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 11–2.

Figure 11–2 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, use the following steps to create dimensions:

1. Identify all dimensions and dimension tables in the schema
2. Identify the hierarchies within each dimension
3. Identify the attribute dependencies within each level of the hierarchy
4. Identify joins from the fact table in a data warehouse to each dimension

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.

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.

Identify the hierarchies within each dimension As an example, day is a child of month (you can aggregate day level data up to month), and quarter is a child of year.

Identify the attribute dependencies within each level of the hierarchy As an example, identify that calendar_month_name is an attribute of month.

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 Dimensions

```
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);
```

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 you 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.

This section contains the following topics:

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 11–1 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 11–2 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 11–3 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 11–4 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 11–5 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 11–6 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 11–7 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 11–8 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 11–9 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 7-11 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 you 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, you can rewrite queries that request data from only the fresh portions of the materialized view. Note that because 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.

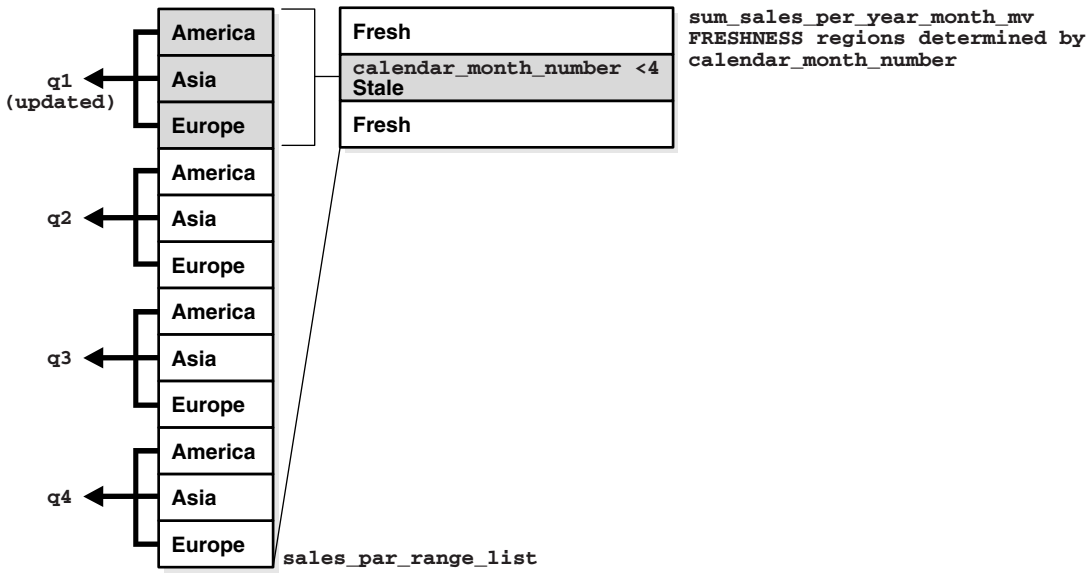
```
SELECT s.calendar_year, SUM(s.amount_sold) AS sum_sales, COUNT(*) AS cnt
FROM sales_par_range_list s
WHERE s.calendar_year = 2000
  AND s.calendar_month_number BETWEEN 2 AND 6
GROUP BY s.calendar_year;
```

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 11–3 offers a graphical illustration of what is stale and what is fresh.

Figure 11–3 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.

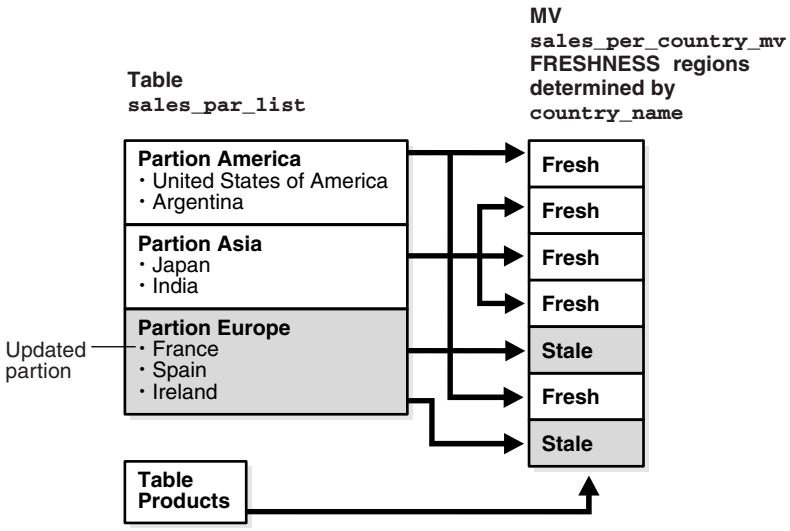
The following updates sales_par_list as follows:

```
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 11–4 illustrates what is fresh and what is stale in this example.

Figure 11–4 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 11–2 lists the partition names and their pmarkers for this example.

Table 11–2 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;
```

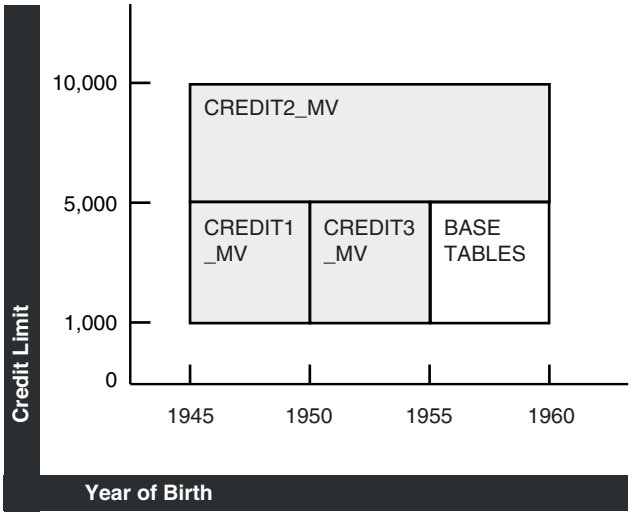
Query 4: Requesting Data on Multiple Selection Columns

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 11–5 shows a two-selection query, which can be rewritten with the two-selection materialized views described in the following section.

Figure 11–5 Query Rewrite Using Multiple Materialized Views



The graph in Figure 11–5 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:

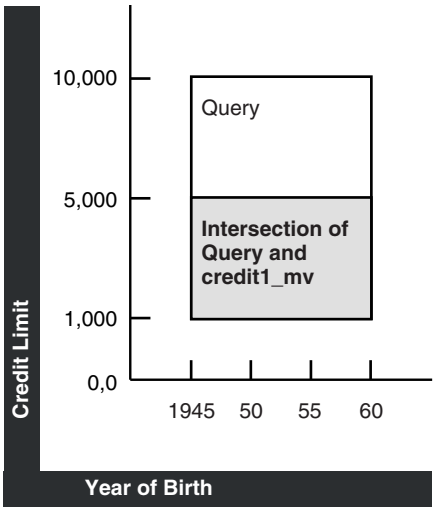
```
SELECT cust_last_name, cust_first_name
FROM credit_mv1
UNION ALL
SELECT cust_last_name, cust_first_name
FROM credit_mv2
UNION ALL
SELECT cust_last_name, cust_first_name
FROM credit_mv3
UNION ALL
SELECT cust_last_name, cust_first_name
FROM customers
WHERE cust_credit_limit BETWEEN 1000 AND 5000
  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 11–6 Constrained Materialized View Selections



The area between the lines in Figure 11–6 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

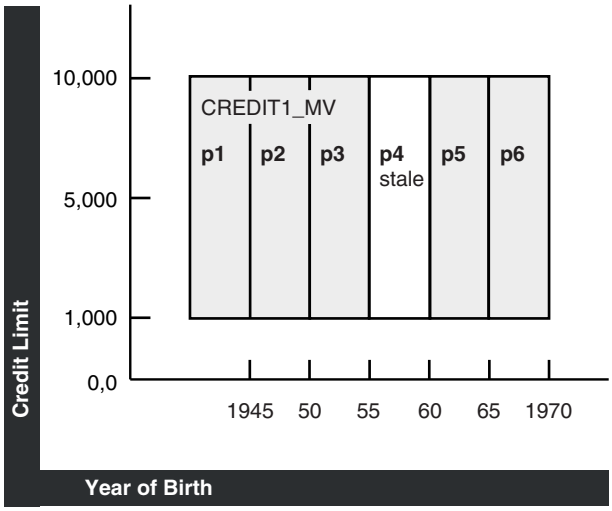
tables to answer the query. Therefore, all the PCT rules and conditions apply here as well. The materialized view should be PCT enabled and the changes made to the base table should be such that the fresh and stale portions of the materialized view can be clearly identified.

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 11–7, 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 11–7 PCT and Multiple Materialized View Rewrite



Let us say that you 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, you 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.

See Also: View Constraints Restrictions

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 11–3 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 11–3 View Constraints and Rewrite Integrity Modes

Constraint States	RELY	NORELY
`ENFORCED`	No	No
`TRUSTED`	Yes	No
`STALE_TOLERATED`	Yes	No

Example 11–10 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 11–3, 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.

See UNION ALL Marker.

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 the following considerations for using query rewrite with grouping sets:

- Query Rewrite When Using GROUP BY Extensions
- Hint for Queries with Extended GROUP BY

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 10-2. 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 11-51 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 10-2. 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 SQL 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 6, "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 you can 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 11–11 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 11–12.

Example 11–12 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 11–13 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 21, "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 11–14 Result Cache Materialized View

First, grant the requisite permissions:

```
CONNECT / AS SYSDBA
GRANT CREATE MATERIALIZED VIEW TO sh;
GRANT EXECUTE ON DBMS_ADVANCED_REWRITE TO sh;
```

Next, 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, 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, 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, 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, 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 SQL 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.

This section contains the following topics:

- Using EXPLAIN PLAN with Query Rewrite
- Using the EXPLAIN_REWRITE Procedure with Query Rewrite

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 Database 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.

This section contains the following topics:

- DBMS_MVIEW.EXPLAIN_REWRITE Syntax
- Using REWRITE_TABLE
- Using a Varray
- EXPLAIN_REWRITE Benefit Statistics
- Support for Query Text Larger than 32KB in EXPLAIN_REWRITE
- EXPLAIN_REWRITE and Multiple Materialized Views
- EXPLAIN_REWRITE Output

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 11–15 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 11-66 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 11–16 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;
```

You 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. Because 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 5-22 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 11-51 for further information.

12

Attribute Clustering

Attribute clustering is a table-level directive that clusters data in close physical proximity based on the content of certain columns. Storing data that logically belongs together in close physical proximity can greatly reduce the amount of data to be processed and can lead to better performance of certain queries in the workload.

This chapter includes the following sections:

About Attribute Clustering

An attribute-clustered table stores data in close proximity on disk in an ordered way based on the values of a certain set of columns in the table or a set of columns in the other tables.

You can cluster according to the linear order of specified columns or by using a function that permits multi-dimensional clustering (also known as interleaved clustering). Attribute clustering improves the effectiveness of zone maps, Exadata Storage Indexes, and In-memory min/max pruning. Queries that qualify clustered columns will access only the clustered regions. When attribute clustering is defined on a partitioned table, the clustering applies to all partitions.

Attribute clustering is a directive property of a table. It is not enforced for every DML operation, but only affects direct-path insert operations, data movement, or table creation. Conventional DML operations on the table are not affected by attribute clustering. This means that whatever is done to cluster the data is an operation that is only done on the current working data set. This is in contrast to a manually-applied `ORDER BY` command, such as what occurs as part of a CTAS operation.

You can cluster data using the following methods:

- Clustering based on one or more columns of the table on which attribute clustering is defined.
- Clustering based on one or more columns that are joined with the table on which attribute clustering is defined. Clustering based on joined columns is called **join attribute clustering**. The tables should be connected through a primary key-foreign key relationship but foreign keys do not have to be enforced.

  Because star queries typically qualify dimension hierarchies, it can be beneficial if fact tables are clustered based on columns (attributes) of one or more dimension tables. With join attribute clustering, you can join one or more dimension tables

with a fact table and then cluster the fact table data by dimension hierarchy columns. To cluster a fact table on columns from one or more dimension tables, the join to the dimension tables must be on a primary or unique key of the dimension tables. Join attribute clustering in the context of star queries is also known as *hierarchical clustering* because the table data is clustered by dimension hierarchies, each made up of an ordered list of hierarchical columns (for example, the `nation`, `state`, and `city` columns forming a `location` hierarchy).

Note: In contrast with Oracle Table Clusters, join attribute clustered tables do not store data from a group of tables in the same database blocks. For example, consider an attribute clustered table `sales` joined with a dimension table `products`. The `sales` table will only contain rows from the `sales` table, but the ordering of the rows will be based on the values of columns joined from `products` table. The appropriate join will be executed during data movement, direct path insert and CTAS operations.

This section contains the following topics:

- Types of Attribute Clustering
- Advantages of Attribute-Clustered Tables
- About Defining Attribute Clustering for Tables
- About Specifying When Attribute Clustering Must be Performed

Types of Attribute Clustering

Attribute clustering is a user-defined table directive that provides data clustering on one or more columns in a table. The directives can be specified when the table is created or modified.

Oracle Database provides the following types of attribute clustering:

- Attribute Clustering with Linear Ordering
- Attribute Clustering with Interleaved Ordering

Regardless of the type of attribute clustering used, you can either cluster data based on a single table or by joining multiple tables (join attribute clustering).

Attribute Clustering with Linear Ordering

Linear ordering stores the data according to the order of specified columns. This is the default type of clustering. For example, linear ordering on the `(prod_id, channel_id)` columns of the table `SALES` sorts the data by `prod_id` first and then by `channel_id`. The sorted data is stored on disk with the data for clustered columns being in close proximity.

Linear ordering can be defined on single tables or multiple tables that are connected through a primary key-foreign key relationship.

Use the `CLUSTERING ... BY LINEAR ORDER` directive to perform attribute clustering based on the order of specified columns.

Attribute clustering based on linear ordering of columns is best used in the following scenarios:

- Queries specify the prefix of columns included in the `CLUSTERING` clause in a single table

For example, if queries on `sales` often specify either a customer ID or a combination of customer ID and product ID, then you could cluster data in the table using the column order `cust_id`, `prod_id`.

- Columns used in the `CLUSTERING` clause have an acceptable level of cardinality

  The potential data reduction that can be obtained in the scenarios described in "Advantages of Attribute-Clustered Tables" on page 12-4 increases in direct proportion to the data reduction obtained from a predicate on a column.

Linear clustering combined with zone maps is very effective in I/O reduction.

Attribute Clustering with Interleaved Ordering

Interleaved ordering uses a special multidimensional clustering technique based on Z-order curve fitting. It maps multiple column attribute values (multidimensional data points) to a single one-dimensional value while preserving the multidimensional locality of column values (data points). Interleaved ordering is supported on single tables or multiple tables. Unlike linear ordering, this method does not require the leading columns of the clustering definition to be present to achieve I/O pruning benefits for the scenarios described in "Advantages of Attribute-Clustered Tables" on page 12-4.

Columns can be used individually or grouped together into column groups. Each individual column or column group will be used to constitute one of the multidimensional data points in the cluster. Grouped columns are bracketed by '('..')', and must follow the dimensional hierarchy from the coarsest to the finest level of granularity. For example, (`product_category`, `product_subcategory`).

Use the `CLUSTERING . . . BY INTERLEAVED ORDER` directive to perform clustering by interleaved ordering.

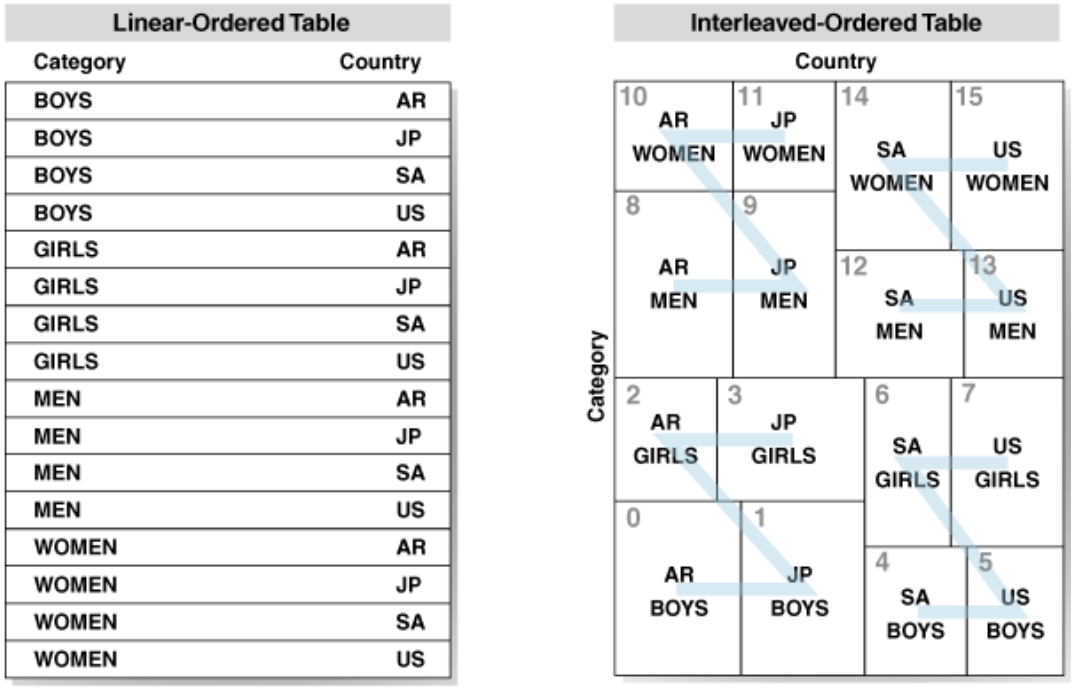
Interleaved clustering is most beneficial for SQL operations with varying predicates on multiple columns. This is often the case for star queries against a dimensional model, where the query predicates are on dimension tables and the number of predicates vary. Using interleaved join attribute clustering is most common in environments where the fact table is clustered based on columns from the dimension tables. The columns from a dimension table will likely contain a hierarchy, for example, the hierarchy of a product category and sub-category. In this case, clustering of the fact table would occur on dimension columns forming a hierarchy. This is the reason join attribute clustering for star schemas is sometimes referred to as hierarchical clustering. For example, if queries on `sales` specify columns from different dimensions, then you could cluster data in the `sales` table according to columns in these dimensions.

Interleaved clustering combined with zone maps is very effective in I/O pruning for star schema queries. In addition, it enables you to provide a very efficient I/O pruning for queries using zone maps, and enhances compression because the same column values are close to each other and can be easily compressed.

Example: Attribute Clustered Table

An example of how a clustered table looks is illustrated in Figure 12–1. Assume you have a table `sales` with columns `(category, country)`. The table on the left is clustered using linear ordering, and the table on the right is clustered using interleaved ordering. Observe that, in the interleaved-ordered table, there are contiguous regions on disk that contain data with a given category and country.

Figure 12–1 Attribute-Clustered Tables



Guidelines for Using Attribute Clustering

The following are some considerations when defining an attribute clustered table:

- Use attribute clustering in combination with zone maps to facilitate zone pruning and its associated I/O reduction.
- Consider large tables that are frequently queried with predicates on medium to low cardinality columns.
- Consider fact tables that are frequently queried by dimensional hierarchies.
- For a partitioned table, consider including columns that correlate with partition keys (to facilitate zone map partition pruning).
- For linear ordering, list columns in prefix-to-suffix order.
- Group together columns that form a dimensional hierarchy. This constitutes a column group. Within each column group, list columns in order of coarsest to finest granularity.
- If there are more than four dimension tables, include the dimensions that are most commonly specified with filters. Limit the number of dimensions to two or three for better clustering effect.
- Consider using attribute clustering instead of indexes on low to medium cardinality columns.
- If the primary key of a dimension table is composed of dimension hierarchy values (for example, the primary key is made up of year, quarter, month, day values), make the corresponding foreign key as clustering column instead of dimension hierarchy.

Advantages of Attribute-Clustered Tables

- Eliminates storage costs associated with using indexes

- Enables the accessing of clustered regions rather than performing random I/O or full table scans when used in conjunction with zone maps
- Provides I/O reduction when used in conjunction with any of the following:
  - Oracle Exadata Storage Indexes
  - Oracle In-memory min/max pruning
  - Zone maps

  Attribute clustering provides data clustering based on the attributes that are used as filter predicates. Because both Exadata Storage Indexes and Oracle In-memory min/max pruning track the minimum and maximum values of columns stored in each physical region, clustering reduces the I/O required to access data.

  I/O pruning using zone maps can significantly reduce I/O costs and CPU cost of table scans and index scans.
- Enables clustering of fact tables based on dimension columns in star schemas

  Techniques such as traditional table clusters do not provide for ordering by columns of other tables. In star schemas, most queries qualify dimension tables and not fact tables, so clustering by fact table columns is not effective. Oracle Database supports clustering on columns in dimension tables.
- Improves data compression ratios and in this way indirectly improves table scan costs

  Compression can be improved because, with clustering, there is a high probability that clustered columns with the same values are close to each other on disk, hence the database can more easily compress them.
- Minimizes table lookup and single block I/O operations for index range scan operations when the attribute clustering is on the index selection criteria.
- Enables I/O reduction in OLTP applications for queries that qualify a prefix in and use attribute clustering with linear order
- Enables I/O reduction on a subset of the clustering columns for attribute clustering with interleaved ordering

  If table data is ordered on multiple columns, as in an index-organized table, then a query must specify a prefix of the columns to gain I/O savings. In contrast, a `BY INTERLEAVED` table permits queries to benefit from I/O pruning when they specify columns from multiple tables in a non-prefix order.

About Defining Attribute Clustering for Tables

Attribute clustering information is part of the table metadata. You can define attribute clustering for a table either when table is first created or subsequently, by altering the table definition.

Use the `CLUSTERTING` clause of the `CREATE TABLE` statement to define attribute clustering for a table. The type of attribute clustering is specified by including `BY LINEAR ORDER` or `BY INTERLEAVED ORDER`.

See Also:

- "Creating Attribute-Clustered Tables with Linear Ordering" on page 12-7
- "Creating Attribute-Clustered Tables with Interleaved Ordering" on page 12-8

If attribute clustering was not defined when the table was created, you can modify the table definition and add clustering. Use the `ALTER TABLE ... ADD CLUSTERING` statement to define attribute clustering for an existing table.

> **See Also:** "Adding Attribute Clustering to an Existing Table" on page 12-10

About Specifying When Attribute Clustering Must be Performed

Performing clustering may be expensive because it involves reorganization of the table and clustering data during DML operations. Oracle Database does not enforce the clustering of data on conventional DML, conventional insert, update, and merge.

Clustering can be performed in two ways. The first is to automatically perform clustering for certain DML operations on the table. This is done by defining, as part of the table metadata, the operations for which clustering is triggered. The second is to explicitly specify that must be performed as described in "Using Hints to Control Attribute Clustering for DML Operations" on page 12-11 and "Overriding Table-level Settings for Attribute Clustering During DDL Operations" on page 12-11. In this case, you can perform clustering for a table even if its metadata definition does not include clustering.

As part of the table definition, you can specify that attribute clustering must be performed when the following operations are triggered:

- Direct-path insert operations

  Set the `ON LOAD` option to `YES` to specify that attribute clustering must be performed during direct-path insert operations. This includes `MERGE` operations with implied direct loads using hints.

- Data movement operations

  Set the `ON DATA MOVEMENT` option to `YES` to specify clustering must be performed during data movement operations. This includes online table redefinition and the following partition operations: `MOVE`, `MERGE`, `SPLIT`, and `COALESCE`.

The `ON LOAD` and `ON DATA MOVEMENT` options can be included in a `CREATE TABLE` or `ALTER TABLE` statement. If neither `YES ON LOAD` nor `YES ON DATA MOVEMENT` is specified, then clustering is not enforced automatically.

It will serve only as metadata defining natural clustering of the table that may be used later for zone map creation. In this case, it is up to the user to enforce clustering during loads.

> **See Also:** "Adding Attribute Clustering to an Existing Table" on page 12-10 for an example on using the `ON LOAD` and `ON DATA MOVEMENT` options

Attribute Clustering Operations

This section describes common tasks involving attribute clustering and includes:

- Privileges for Attribute-Clustered Tables
- Creating Attribute-Clustered Tables with Linear Ordering
- Creating Attribute-Clustered Tables with Interleaved Ordering
- Maintaining Attribute Clustering