Oracle® Database

VLDB and Partitioning Guide

11*g* Release 1 (11.1)

B32024-01

July 2007

Oracle Database VLDB and Partitioning Guide, 11*g* Release 1 (11.1)

B32024-01

Primary Author: Tony Morales

Contributors: Hermann Baer, Steve Fogel, Lilian Hobbs, Paul Lane, Diana Lorentz, Valarie Moore, Mark Van de Wiel

Contents

Preface

This book contains an overview of very large database (VLDB) topics, with emphasis on partitioning as a key component of the VLDB strategy. Partitioning enhances the performance, manageability, and availability of a wide variety of applications and helps reduce the total cost of ownership for storing large amounts of data.

Audience

This document is intended for DBAs and developers who create, manage, and write applications for very large databases (VLDB).

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see the following documents in the Oracle Database documentation set:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database SQL Language Reference*
- *Oracle Database Data Warehousing Guide*
- *Oracle Database Performance Tuning Guide*

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.

1

Introduction to Very Large Databases

Modern enterprises frequently run mission-critical databases containing upwards of several hundred gigabytes and, in many cases, several terabytes of data. These enterprises are challenged by the support and maintenance requirements of very large databases (VLDB), and must devise methods to meet those challenges.

This chapter contains an overview of VLDB topics, with emphasis on partitioning as a key component of the VLDB strategy.

- Introduction to Partitioning
- VLDB and Partitioning
- Partitioning As the Foundation for Information Lifecycle Management
- Partitioning for Every Database

Note: Partitioning functionality is available only if you purchase the Partitioning option.

Introduction to Partitioning

Partitioning addresses key issues in supporting very large tables and indexes by letting you decompose them into smaller and more manageable pieces called **partitions**, which are entirely transparent to an application. SQL queries and DML statements do not need to be modified in order to access partitioned tables. However, after partitions are defined, DDL statements can access and manipulate individual partitions rather than entire tables or indexes. This is how partitioning can simplify the manageability of large database objects.

Each partition of a table or index must have the same logical attributes, such as column names, datatypes, and constraints, but each partition can have separate physical attributes such as compression enabled or disabled, physical storage settings, and tablespaces.

Partitioning is useful for many different types of applications, particularly applications that manage large volumes of data. OLTP systems often benefit from improvements in manageability and availability, while data warehousing systems benefit from performance and manageability.

Partitioning offers these advantages:

- It enables data management operations such as data loads, index creation and rebuilding, and backup/recovery at the partition level, rather than on the entire table. This results in significantly reduced times for these operations.

- It improves query performance. In many cases, the results of a query can be achieved by accessing a subset of partitions, rather than the entire table. For some queries, this technique (called **partition pruning**) can provide order-of-magnitude gains in performance.
- It significantly reduces the impact of scheduled downtime for maintenance operations.

  Partition independence for partition maintenance operations lets you perform concurrent maintenance operations on different partitions of the same table or index. You can also run concurrent `SELECT` and DML operations against partitions that are unaffected by maintenance operations.
- It increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures.
- Parallel execution provides specific advantages to optimize resource utilization, and minimize execution time. Parallel execution against partitioned objects is key for scalability in a clustered environment. Parallel execution is supported for queries as well as for DML and DDL.

Partitioning enables faster data access within an Oracle database. Whether a database has 10 GB or 10 TB of data, partitioning can improve data access by orders of magnitude. Partitioning can be implemented without requiring any modifications to your applications. For example, you could convert a nonpartitioned table to a partitioned table without needing to modify any of the `SELECT` statements or DML statements which access that table. You do not need to rewrite your application code to take advantage of partitioning.

VLDB and Partitioning

A very large database has no minimum absolute size. Although a VLDB is a database like smaller databases, there are specific challenges in managing a VLDB. These challenges are related to the sheer size, and the cost-effectiveness of performing operations against a system that size, taken for granted on smaller databases.

Several trends have been responsible for the steady growth in database size:

- For a long time, systems have been developed in isolation. Companies have started to see the benefits of combining these systems to enable cross-departmental analysis while reducing system maintenance costs. Consolidation of databases and applications is a key factor in the ongoing growth of database size.
- Many companies face regulations that set specific requirements for storing data for a minimum amount of time. The regulations generally result in more data being stored for longer periods of time.
- Companies grow organically and through mergers and acquisitions, causing the amount of generated and processed data to increase. At the same time, the user population that relies on the database for daily activities increases.

Partitioning is a critical feature for managing very large databases. Growth is the basic challenge that partitioning addresses for very large databases, and partitioning enables a "divide and conquer" technique for managing the tables and indexes in the database, especially as those tables and indexes grow. Partitioning is the feature that allows a database to scale for very large datasets while maintaining consistent performance, without unduly increasing administrative or hardware resources. Chapter 4 provides availability, manageability, and performance considerations for partitioning implementations.

Chapter 8 addresses the challenges surrounding backup and recovery for a VLDB.

Storage is a key component of a very large database. Chapter 9 focuses on best practices for storage in a VLDB.

Partitioning As the Foundation for Information Lifecycle Management

Information Lifecycle Management (ILM) is a set of processes and policies for managing data throughout its useful life. One important component of an ILM strategy is determining the most appropriate and cost-effective medium for storing data at any point during its life time: newer data used in day-to-day operations is stored on the fastest, most highly-available storage tier, while older data which is accessed infrequently may be stored on a less-expensive and less-performant storage tier. Older data may also be updated less frequently in which case it makes sense to compress and store the data as read-only.

Oracle Database provides the ideal environment for implementing your ILM solution. Oracle supports multiple storage tiers, and since all of the data remains in the Oracle database, the use of multiple storage tiers is completely transparent to the application and the data continues to be completely secure. Partitioning provides the fundamental technology that enables data in tables to be stored in different partitions.

Although multiple storage tiers and sophisticated ILM policies are most often found in enterprise-level systems, most companies and most databases need some degree of information lifecycle management. The most basic of ILM operations, archiving older data and purging or removing that data from the database, can be orders of magnitude faster when using partitioning.

See Also: Chapter 5, "Using Partitioning for Information Lifecycle Management" for more details on ILM

Partitioning for Every Database

The benefits of partitioning are not just for very large databases; every database, even small databases, can benefit from partitioning. While partitioning is a necessity for the largest databases in the world, partitioning is obviously beneficial for the smaller database as well. Even a database whose size is measured in megabytes will see the same type of performance and manageability benefits from partitioning as the largest multi-terabyte systems.

See Also:

- Chapter 6, "Using Partitioning in a Data Warehouse Environment" for more details on how partitioning can provide benefits in a data warehouse environment
- Chapter 7, "Using Partitioning in an Online Transaction Processing Environment" for more details on how partitioning can provide benefits in an OLTP environment

2 Partitioning Concepts

Partitioning enhances the performance, manageability, and availability of a wide variety of applications and helps reduce the total cost of ownership for storing large amounts of data. Partitioning allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity. Oracle provides a rich variety of partitioning strategies and extensions to address every business requirement. Moreover, since it is entirely transparent, partitioning can be applied to almost any application without the need for potentially expensive and time consuming application changes.

This chapter contains the following topics:

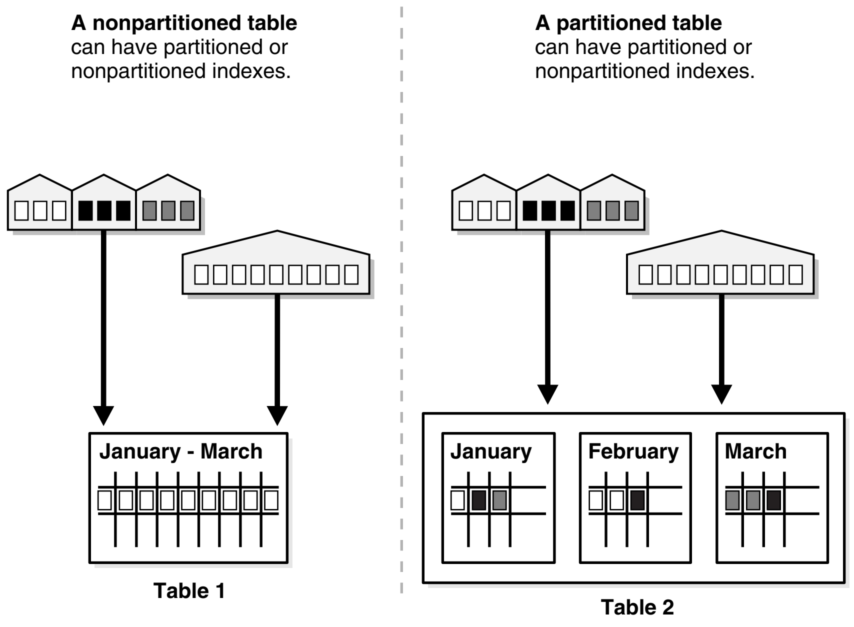
Basics of Partitioning

Partitioning allows a table, index, or index-organized table to be subdivided into smaller pieces, where each piece of such a 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 queries and DML statements.

Figure 2–1 offers a graphical view of how partitioned tables differ from non-partitioned tables.

Figure 2–1 A View of Partitioned Tables



Note: All partitions of a partitioned object must reside in tablespaces of a single block size.

See Also: *Oracle Database Concepts* for more information about multiple block sizes

Partitioning Key

Each row in a partitioned table is unambiguously assigned to a single partition. The partitioning key is comprised of one or more columns that determine the partition where each row will be stored. Oracle automatically directs insert, update, and delete operations to the appropriate partition through the use of the partitioning key.

Partitioned Tables

Any table can be partitioned into a million separate partitions except those tables containing columns with `LONG` or `LONG RAW` datatypes. You can, however, use tables containing columns with `CLOB` or `BLOB` datatypes.

Note: To reduce disk usage and memory usage (specifically, the buffer cache), you can store tables and partitions of a partitioned table in a compressed format inside the database. This often leads to a better scaleup for read-only operations. Table compression can also speed up query execution. There is, however, a slight cost in CPU overhead.

See Also: *Oracle Database Concepts* for more information about table compression

When to Partition a Table

Here are some suggestions for when to partition a table:

- Tables greater than 2 GB should always be considered as candidates for partitioning.
- Tables containing historical data, in which new data is added into the newest partition. A typical example is a historical table where only the current month's data is updatable and the other 11 months are read only.
- When the contents of a table need to be distributed across different types of storage devices.

When to Partition an Index

Here are some suggestions for when to consider partitioning an index:

- Avoid rebuilding the entire index when data is removed.
- Perform maintenance on parts of the data without invalidating the entire index.
- Reduce the impact of index skew caused by an index on a column with a monotonically increasing value.

Partitioned Index-Organized Tables

Partitioned index-organized tables are very useful for providing improved performance, manageability, and availability for index-organized tables.

For partitioning an index-organized table:

- Partition columns must be a subset of the primary key columns
- Secondary indexes can be partitioned (both locally and globally)
- `OVERFLOW` data segments are always equi-partitioned with the table partitions

> **See Also:** *Oracle Database Concepts* for more information about index-organized tables

System Partitioning

System partitioning enables application-controlled partitioning without having the database controlling the data placement. The database simply provides the ability to break down a table into partitions without knowing what the individual partitions are going to be used for. All aspects of partitioning have to be controlled by the application. For example, an insertion into a system partitioned table without the explicit specification of a partition will fail.

System partitioning provides the well-known benefits of partitioning (scalability, availability, and manageability), but the partitioning and actual data placement are controlled by the application.

> **See Also:** *Oracle Database Data Cartridge Developer's Guide* for more information about system partitioning

Partitioning for Information Lifecycle Management

Information Lifecycle Management (ILM) is concerned with managing data during its lifetime. Partitioning plays a key role in ILM because it enables groups of data (that is, partitions) to be distributed across different types of storage devices and managed individually.

See Also: Chapter 5, "Using Partitioning for Information Lifecycle Management" for more information about Information Lifecycle Management

Partitioning and LOB Data

Unstructured data (such as images and documents) which is stored in a LOB column in the database can also be partitioned. When a table is partitioned, all the columns will reside in the tablespace for that partition, with the exception of LOB columns, which can be stored in their own tablespace.

This technique is very useful when a table is comprised of large LOBs because they can be stored separately from the main data. This can be beneficial if the main data is being frequently updated but the LOB data isn't. For example, an employee record may contain a photo which is unlikely to change frequently. However, the employee personnel details (such as address, department, manager, and so on) could change. This approach also means that cheaper storage can be used for storing the LOB data and more expensive, faster storage used for the employee record.

Benefits of Partitioning

Partitioning can provide tremendous benefit to a wide variety of applications by improving performance, manageability, and availability. It is not unusual for partitioning to improve the performance of certain queries or maintenance operations by an order of magnitude. Moreover, partitioning can greatly simplify common administration tasks.

Partitioning also enables database designers and administrators to tackle some of the toughest problems posed by cutting-edge applications. Partitioning is a key tool for building multi-terabyte systems or systems with extremely high availability requirements.

Partitioning for Performance

By limiting the amount of data to be examined or operated on, and by providing data distribution for parallel execution, partitioning provides a number of performance benefits. These features include:

- Partition Pruning
- Partition-Wise Joins

Partition Pruning

Partition 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 a historical record of orders, and that this table has been partitioned by week. A query requesting orders for a single week would only access a single partition of the `Orders` table. If the `Orders` table had 2 years of historical data, then this query would access one partition instead of 104 partitions. This query could potentially execute 100 times faster simply because of partition pruning.

Partition pruning works with all of Oracle's other performance features. Oracle 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 both tables are partitioned on the join key, or when a reference partitioned table is joined with its parent table. Partition-wise joins break a large join into smaller joins that occur between each of the partitions, completing the overall join in less time. This offers significant performance benefits both for serial and parallel execution.

Partitioning for Manageability

Partitioning allows tables and indexes to be partitioned into smaller, more manageable units, providing database administrators with the ability to pursue a "divide and conquer" approach to data management. With partitioning, maintenance operations can be focused on particular portions of tables. For example, a database administrator could back up a single partition of a table, rather than backing up the entire table. For maintenance operations across an entire database object, it is possible to perform these operations on a per-partition basis, thus dividing the maintenance process into more manageable chunks.

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 weekly basis. That table could be partitioned so that each partition contains one week of data. The load process is simply the addition of a new partition using a partition exchange load. Adding a single partition is much more efficient than modifying the entire table, since the DBA does not need to modify any other partitions.

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, then all of the other partitions of the table remain online and available. The application can continue to execute queries and transactions against the available partitions for the table, and these database operations will run successfully, provided they do not need to access the unavailable partition.

The database administrator can specify that each partition be stored in a separate tablespace; the most common scenario is having these tablespaces stored on different storage tiers. Storing different partitions in different tablespaces allows the database administrator to do backup and recovery operations on each individual partition, independent of the other partitions in the table. Thus allowing the active parts of the database to be made available sooner so access to the system can continue, while the inactive data is still being restored. Moreover, partitioning can reduce scheduled downtime. The performance gains provided by partitioning may enable database administrators to complete maintenance operations on large database objects in relatively small batch windows.

Partitioning Strategies

Oracle Partitioning offers three fundamental data distribution methods as basic partitioning strategies that control how data is placed into individual partitions:

- Range
- Hash

- List

Using these data distribution methods, a table can either be partitioned as a single list or as a composite partitioned table:

- Single-Level Partitioning
- Composite Partitioning

Each partitioning strategy has different advantages and design considerations. Thus, each strategy is more appropriate for a particular situation.

Single-Level Partitioning

A table is defined by specifying one of the following data distribution methodologies, using one or more columns as the partitioning key:

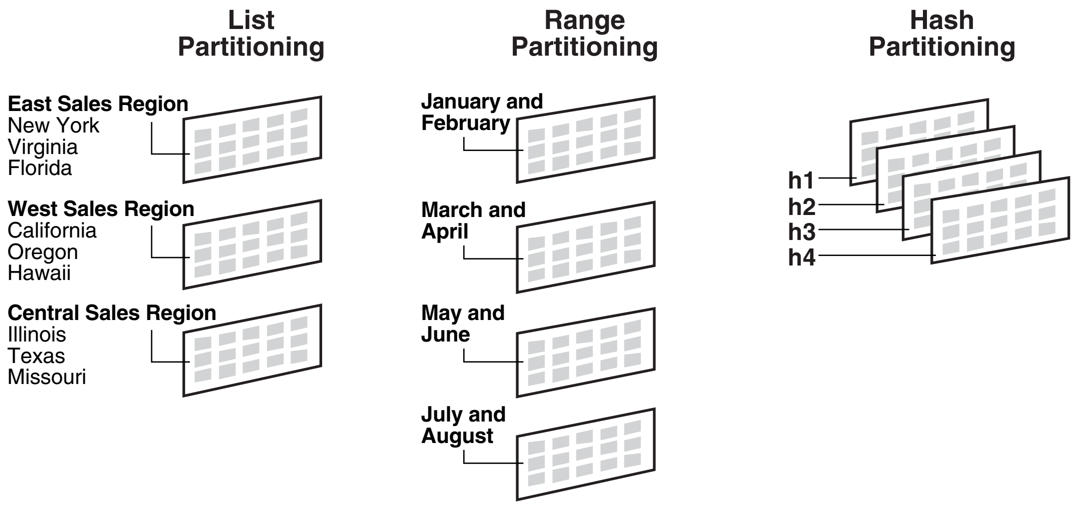
- Range Partitioning
- Hash Partitioning
- List Partitioning

For example, consider a table with a column of type `NUMBER` as the partitioning key and two partitions `less_than_five_hundred` and `less_than_one_thousand`. The `less_than_one_thousand` partition contains rows where the following condition is true:

```
500 <= partitioning key < 1000
```

Figure 2–2 offers a graphical view of the basic partitioning strategies for a single-level partitioned table.

Figure 2–2 List, Range, and Hash Partitioning



Range Partitioning

Range partitioning maps data to partitions based on ranges of values of the partitioning key that you establish for each partition. It is the most common type of partitioning and is often used with dates. For a table with a date column as the partitioning key, the `January-2005` partition would contain rows with partitioning key values from 01-Jan-2005 to 31-Jan-2005.

Each partition has a `VALUES LESS THAN` clause, which specifies a non-inclusive upper bound for the partitions. Any values of the partitioning key equal to or higher than this literal are added to the next higher partition. All partitions, except the first, have an implicit lower bound specified by the `VALUES LESS THAN` clause of the previous partition.

A `MAXVALUE` literal can be defined for the highest partition. `MAXVALUE` represents a virtual infinite value that sorts higher than any other possible value for the partitioning key, including the NULL value.

Hash Partitioning

Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify. The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size.

Hash partitioning is the ideal method for distributing data evenly across devices. Hash partitioning is also an easy-to-use alternative to range partitioning, especially when the data to be partitioned is not historical or has no obvious partitioning key.

Note: You cannot change the hashing algorithms used by partitioning.

List Partitioning

List partitioning enables you to explicitly control how rows map to partitions by specifying a list of discrete values for the partitioning key in the description for each partition. The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way. For a table with a region column as the partitioning key, the `North America` partition might contain values `Canada`, `USA`, and `Mexico`.

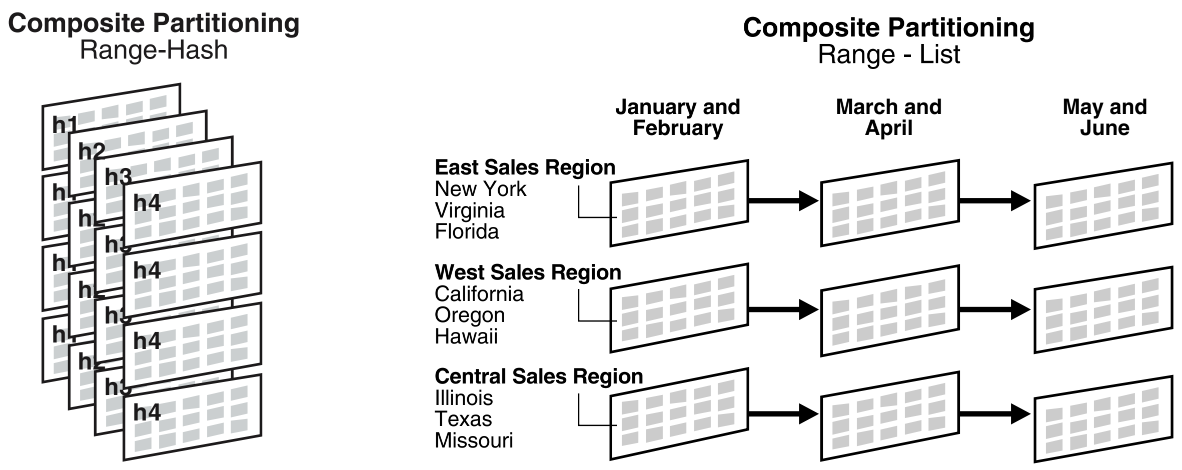
The `DEFAULT` partition enables you to avoid specifying all possible values for a list-partitioned table by using a default partition, so that all rows that do not map to any other partition do not generate an error.

Composite Partitioning

Composite partitioning is a combination of the basic data distribution methods; a table is partitioned by one data distribution method and then each partition is further subdivided into subpartitions using a second data distribution method. All subpartitions for a given partition together represent a logical subset of the data.

Composite partitioning supports historical operations, such as adding new range partitions, but also provides higher degrees of potential partition pruning and finer granularity of data placement through subpartitioning. Figure 2–3 offers a graphical view of range-hash and range-list composite partitioning, as an example.

Figure 2–3 Composite Partitioning



- Composite Range-Range Partitioning
- Composite Range-Hash Partitioning
- Composite Range-List Partitioning
- Composite List-Range Partitioning
- Composite List-Hash Partitioning
- Composite List-List Partitioning

Composite Range-Range Partitioning

Composite range-range partitioning enables logical range partitioning along two dimensions; for example, partition by `order_date` and range subpartition by `shipping_date`.

Composite Range-Hash Partitioning

Composite range-hash partitioning partitions data using the range method, and within each partition, subpartitions it using the hash method. Composite range-hash partitioning provides the improved manageability of range partitioning and the data placement, striping, and parallelism advantages of hash partitioning.

Composite Range-List Partitioning

Composite range-list partitioning partitions data using the range method, and within each partition, subpartitions it using the list method. Composite range-list partitioning provides the manageability of range partitioning and the explicit control of list partitioning for the subpartitions.

Composite List-Range Partitioning

Composite list-range partitioning enables logical range subpartitioning within a given list partitioning strategy; for example, list partition by `country_id` and range subpartition by `order_date`.

Composite List-Hash Partitioning

Composite list-hash partitioning enables hash subpartitioning of a list-partitioned object; for example, to enable partition-wise joins.

Composite List-List Partitioning

Composite list-list partitioning enables logical list partitioning along two dimensions; for example, list partition by `country_id` and list subpartition by `sales_channel`.

Partitioning Extensions

In addition to the basic partitioning strategies, Oracle Database provides partitioning extensions:

- Manageability Extensions
- Partitioning Key Extensions

Manageability Extensions

These extensions significantly enhance the manageability of partitioned tables:

- Interval Partitioning
- Partition Advisor

Interval Partitioning

Interval partitioning is an extension of range partitioning which instructs the database to automatically create partitions of a specified interval when data inserted into the table exceeds all of the existing range partitions. You must specify at least one range partition. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database creates interval partitions for data beyond that transition point. The lower boundary of every interval partition is the non-inclusive upper boundary of the previous range or interval partition.

For example, if you create an interval partitioned table with monthly intervals and the transition point at January 1, 2007, then the lower boundary for the January 2007 interval is January 1, 2007. The lower boundary for the July 2007 interval is July 1, 2007, regardless of whether the June 2007 partition was already created.

When using interval partitioning, consider the following restrictions:

- You can only specify one partitioning key column, and it must be of `NUMBER` or `DATE` type.
- Interval partitioning is not supported for index-organized tables.
- You cannot create a domain index on an interval-partitioned table.

You can create single-level interval partitioned tables as well as the following composite partitioned tables:

- Interval-range
- Interval-hash
- Interval-list

Partition Advisor

The Partition Advisor is part of the SQL Access Advisor. The Partition Advisor can recommend a partitioning strategy for a table based on a supplied workload of SQL statements which can be supplied by the SQL Cache, a SQL Tuning set, or be defined by the user.

Partitioning Key Extensions

These extensions extend the flexibility in defining partitioning keys:

- Reference Partitioning
- Virtual Column-Based Partitioning

Reference Partitioning

Reference partitioning allows the partitioning of two tables related to one another by referential constraints. The partitioning key is resolved through an existing parent-child relationship, enforced by enabled and active primary key and foreign key constraints.

The benefit of this extension is that tables with a parent-child relationship can be logically equi-partitioned by inheriting the partitioning key from the parent table without duplicating the key columns. The logical dependency will also automatically cascade partition maintenance operations, thus making application development easier and less error-prone.

An example of reference partitioning is the `Orders` and `OrderItems` tables related to each other by a referential constraint `orderid_refconstraint`. Namely, `OrderItems.OrderID` references `Orders.OrderID`. The `Orders` table is range partitioned on `OrderDate`. Reference partitioning on `orderid_refconstraint` for `OrderItems` leads to creation of the following partitioned table, which is equi-partitioned with respect to the `Orders` table, as shown in Figure 2–4 and Figure 2–5.

Figure 2–4 Before Reference Partitioning

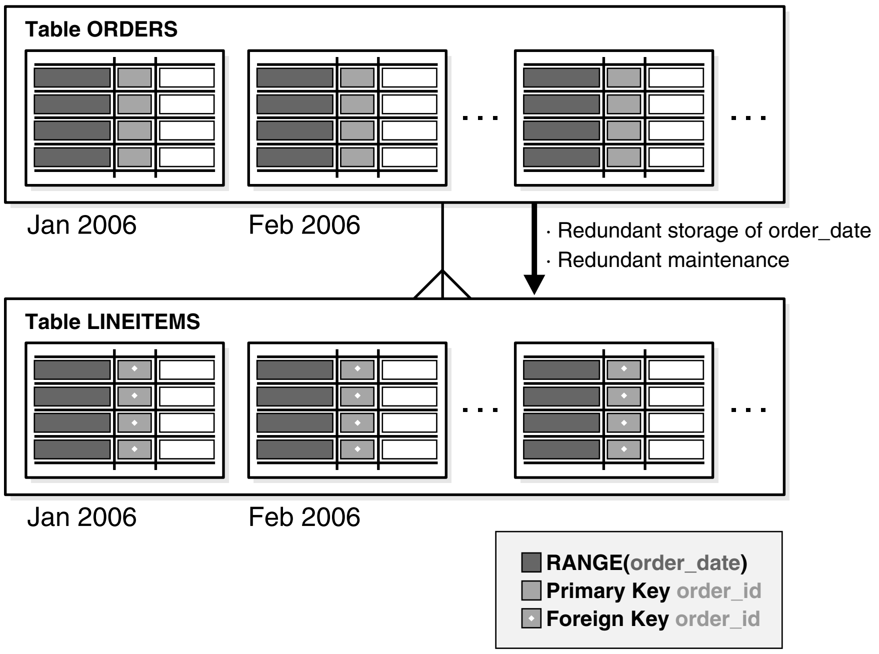
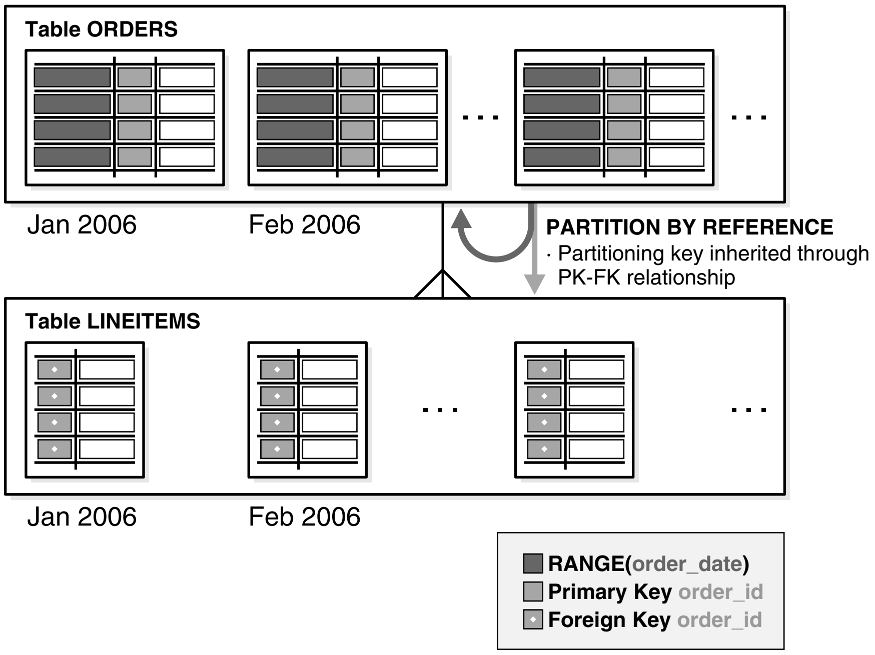


Figure 2–5 With Reference Partitioning



All basic partitioning strategies are available for reference Partitioning. Interval partitioning cannot be used with reference partitioning.

Virtual Column-Based Partitioning

In previous releases of the Oracle Database, a table could only be partitioned if the partitioning key physically existed in the table. In Oracle Database 11*g*, virtual columns remove that restriction and allow the partitioning key to be defined by an expression, using one or more existing columns of a table. The expression is stored as metadata only.

Oracle Partitioning has been enhanced to allow a partitioning strategy to be defined on virtual columns. For example, a 10 digit account ID can include account branch information as the leading 3 digits. With the extension of virtual column based Partitioning, an `ACCOUNTS` table containing an `ACCOUNT_ID` column can be extended with a virtual (derived) column `ACCOUNT_BRANCH` that is derived from the first three digits of the `ACCOUNT_ID` column, which becomes the partitioning key for this table.

Virtual column-based Partitioning is supported with all basic partitioning strategies, including interval and interval-* composite partitioning.

Overview of Partitioned Indexes

Just like partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability. They can either be partitioned independently (global indexes) or automatically linked to a table's partitioning method (local indexes). In general, you should use global indexes for OLTP applications and local indexes for data warehousing or DSS applications. Also, whenever possible, you should try to use local indexes because they are easier to manage. When deciding what kind of partitioned index to use, you should consider the following guidelines in order:

1. If the table partitioning column is a subset of the index keys, use a local index. If this is the case, you are finished. If this is not the case, continue to guideline 2.

2. If the index is unique and does not include the partitioning key columns, then use a global index. If this is the case, then you are finished. Otherwise, continue to guideline 3.
3. If your priority is manageability, use a local index. If this is the case, you are finished. If this is not the case, continue to guideline 4.
4. If the application is an OLTP one and users need quick response times, use a global index. If the application is a DSS one and users are more interested in throughput, use a local index.

See Also: Chapter 6, "Using Partitioning in a Data Warehouse Environment" and Chapter 7, "Using Partitioning in an Online Transaction Processing Environment" for more information about partitioned indexes and how to decide which type to use

Local Partitioned Indexes

Local partitioned indexes are easier to manage than other types of partitioned indexes. They also offer greater availability and are common in DSS environments. The reason for this is equipartitioning: each partition of a local index is associated with exactly one partition of the table. This enables Oracle to automatically keep the index partitions in sync with the table partitions, and makes each table-index pair independent. Any actions that make one partition's data invalid or unavailable only affect a single partition.

Local partitioned indexes support more availability when there are partition or subpartition maintenance operations on the table. A type of index called a local nonprefixed index is very useful for historical databases. In this type of index, the partitioning is not on the left prefix of the index columns.

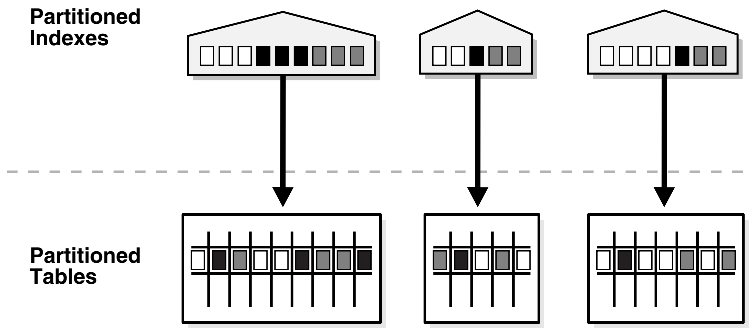
See Also: Chapter 4 for more information about prefixed indexes

You cannot explicitly add a partition to a local index. Instead, new partitions are added to local indexes only when you add a partition to the underlying table. Likewise, you cannot explicitly drop a partition from a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.

A local index can be unique. However, in order for a local index to be unique, the partitioning key of the table must be part of the index’s key columns.

Figure 2–6 offers a graphical view of local partitioned indexes.

Figure 2–6 Local Partitioned Index



Global Partitioned Indexes

Oracle offers two types of global partitioned indexes: range partitioned and hash partitioned.

Global Range Partitioned Indexes

Global range partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method.

The highest partition of a global index must have a partition bound, all of whose values are `MAXVALUE`. This ensures that all rows in the underlying table can be represented in the index. Global prefixed indexes can be unique or nonunique.

You cannot add a partition to a global index because the highest partition always has a partition bound of `MAXVALUE`. If you wish to add a new highest partition, use the `ALTER INDEX SPLIT PARTITION` statement. If a global index partition is empty, you can explicitly drop it by issuing the `ALTER INDEX DROP PARTITION` statement. If a global index partition contains data, dropping the partition causes the next highest partition to be marked unusable. You cannot drop the highest partition in a global index.

Global Hash Partitioned Indexes

Global hash partitioned indexes improve performance by spreading out contention when the index is monotonically growing. In other words, most of the index insertions occur only on the right edge of an index.

Maintenance of Global Partitioned Indexes

By default, the following operations on partitions on a heap-organized table mark all global indexes as unusable:

```
ADD (HASH)
COALESCE (HASH)
DROP
EXCHANGE
MERGE
MOVE
SPLIT
TRUNCATE
```

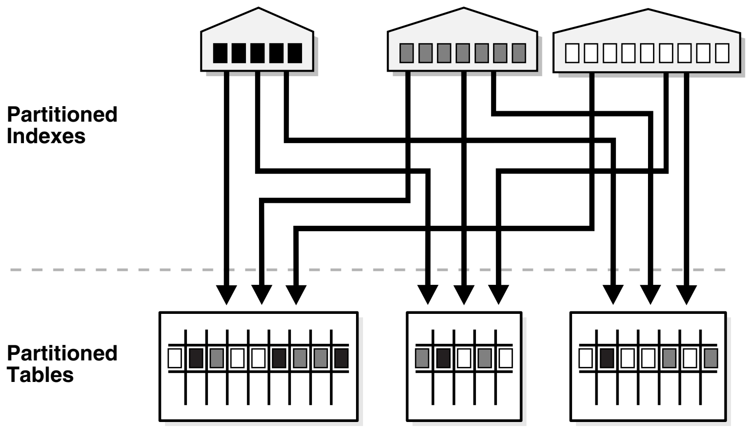
These indexes can be maintained by appending the clause `UPDATE INDEXES` to the SQL statements for the operation. The two advantages to maintaining global indexes:

- The index remains available and online throughout the operation. Hence no other applications are affected by this operation.
- The index doesn't have to be rebuilt after the operation.

Note: This feature is supported only for heap-organized tables.

Figure 2–7 offers a graphical view of global partitioned indexes.

Figure 2–7 Global Partitioned Index

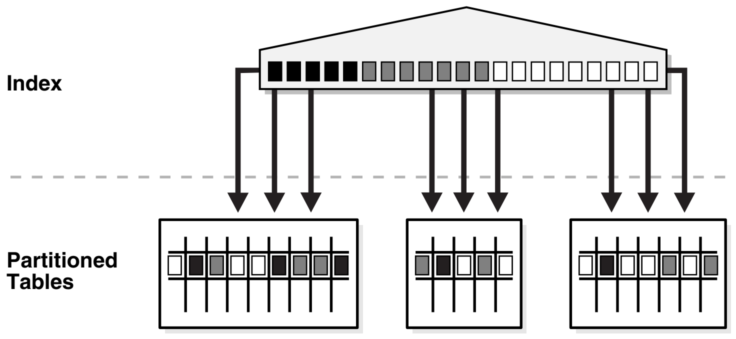


Global Non-Partitioned Indexes

Global non-partitioned indexes behave just like a non-partitioned index.

Figure 2–8 offers a graphical view of global non-partitioned indexes.

Figure 2–8 Global Non-Partitioned Index



Miscellaneous Information about Creating Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables, with the restriction that the bitmap indexes must be local to the partitioned table. They cannot be global indexes.

Global indexes can be unique. Local indexes can only be unique if the partitioning key is a part of the index key.

Partitioned Indexes on Composite Partitions

Here are a few points to remember when using partitioned indexes on composite partitions:

- Subpartitioned indexes are always local and stored with the table subpartition by default.
- Tablespaces can be specified at either index or index subpartition levels.

3 Partition Administration

Partition administration is one of the most important tasks when working with partitioned tables and indexes. This chapter describes various aspects of creating and maintaining partitioned tables and indexes, and contains the following topics:

- Creating Partitions
- Maintaining Partitions
- Dropping Partitioned Tables
- Partitioned Tables and Indexes Example
- Viewing Information About Partitioned Tables and Indexes

Note: Before you attempt to create a partitioned table or index, or perform maintenance operations on any partitioned table, it is recommended that you review the information in Chapter 2, "Partitioning Concepts".

Creating Partitions

Creating a partitioned table or index is very similar to creating a non-partitioned table or index (as described in *Oracle Database Administrator's Guide*), but you include a partitioning clause in the `CREATE TABLE` statement. The partitioning clause, and subclauses, that you include depend upon the type of partitioning you want to achieve.

Partitioning is possible on both regular (heap organized) tables and index-organized tables, except for those containing `LONG` or `LONG RAW` columns. You can create non-partitioned global indexes, range or hash-partitioned global indexes, and local indexes on partitioned tables.

When you create (or alter) a partitioned table, a row movement clause (either `ENABLE ROW MOVEMENT` or `DISABLE ROW MOVEMENT`) can be specified. This clause either enables or disables the migration of a row to a new partition if its key is updated. The default is `DISABLE ROW MOVEMENT`.

The following sections present details and examples of creating partitions for the various types of partitioned tables and indexes:

- Creating Range-Partitioned Tables and Global Indexes
- Creating Interval-Partitioned Tables
- Creating Hash-Partitioned Tables and Global Indexes
- Creating List-Partitioned Tables

- Creating Reference-Partitioned Tables
- Creating Composite Partitioned Tables
- Using Subpartition Templates to Describe Composite Partitioned Tables
- Using Multicolumn Partitioning Keys
- Using Virtual Column-Based Partitioning
- Using Table Compression with Partitioned Tables
- Using Key Compression with Partitioned Indexes
- Creating Partitioned Index-Organized Tables
- Partitioning Restrictions for Multiple Block Sizes

See Also:

- *Oracle Database SQL Language Reference* for the exact syntax of the partitioning clauses for creating and altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables
- *Oracle Database Large Objects Developer's Guide* for information specific to creating partitioned tables containing columns with `LOB`s or other objects stored as `LOB`s
- *Oracle Database Object-Relational Developer's Guide* for information specific to creating tables with object types, nested tables, or VARRAYs

Creating Range-Partitioned Tables and Global Indexes

The `PARTITION BY RANGE` clause of the `CREATE TABLE` statement specifies that the table or index is to be range-partitioned. The `PARTITION` clauses identify the individual partition ranges, and the optional subclauses of a `PARTITION` clause can specify physical and other attributes specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their underlying table.

Creating a Range Partitioned Table

The following example creates a table of four partitions, one for each quarter of sales. The columns `sale_year`, `sale_month`, and `sale_day` are the **partitioning columns**, while their values constitute the **partitioning key** of a specific row. The `VALUES LESS THAN` clause determines the **partition bound**: rows with partitioning key values that compare less than the ordered list of values specified by the clause are stored in the partition. Each partition is given a name (`sales_q1`, `sales_q2`, ...), and each partition is contained in a separate tablespace (`tsa`, `tsb`, ...).

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id)
 ( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
    TABLESPACE tsa
 , PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
    TABLESPACE tsb
 , PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
    TABLESPACE tsc
 , PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
    TABLESPACE tsd
 );
```

A row with `time_id=17-MAR-2006` would be stored in partition `sales_q1_2006`.

> **See Also:** "Using Multicolumn Partitioning Keys" on page 3-20

In the following example, more complexity is added to the example presented earlier for a range-partitioned table. Storage parameters and a `LOGGING` attribute are specified at the table level. These replace the corresponding defaults inherited from the tablespace level for the table itself, and are inherited by the range partitions. However, because there was little business in the first quarter, the storage attributes for partition `sales_q1_2006` are made smaller. The `ENABLE ROW MOVEMENT` clause is specified to allow the automatic migration of a row to a new partition if an update to a key value is made that would place the row in a different partition.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 STORAGE (INITIAL 100K NEXT 50K) LOGGING
 PARTITION BY RANGE (time_id)
 ( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
    TABLESPACE tsa STORAGE (INITIAL 20K NEXT 10K)
 , PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
    TABLESPACE tsb
 , PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
    TABLESPACE tsc
 , PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
    TABLESPACE tsd
 )
 ENABLE ROW MOVEMENT;
```

Creating a Range-Partitioned Global Index

The rules for creating range-partitioned global indexes are similar to those for creating range-partitioned tables. The following is an example of creating a range-partitioned global index on `sale_month` for the tables created in the preceding examples. Each index partition is named but is stored in the default tablespace for the index.

```
CREATE INDEX amount_sold_ix ON sales(amount_sold)
   GLOBAL PARTITION BY RANGE(sale_month)
      ( PARTITION p_100 VALUES LESS THAN (100)
      , PARTITION p_1000 VALUES LESS THAN (1000)
      , PARTITION p_10000 VALUES LESS THAN (10000)
      , PARTITION p_100000 VALUES LESS THAN (100000)
      , PARTITION p_1000000 VALUES LESS THAN (1000000)
      , PARTITION p_greater_than_1000000 VALUES LESS THAN (maxvalue)
      );
```

Note: If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns, because the sort sequence of characters is not identical in all character sets. For more information, see *Oracle Database Globalization Support Guide.*

Creating Interval-Partitioned Tables

The `INTERVAL` clause of the `CREATE TABLE` statement establishes interval partitioning for the table. You must specify at least one range partition using the `PARTITION` clause. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database automatically creates interval partitions for data beyond that transition point. The lower boundary of every interval partition is the non-inclusive upper boundary of the previous range or interval partition.

For example, if you create an interval partitioned table with monthly intervals and the transition point at January 1, 2007, then the lower boundary for the January 2007 interval is January 1, 2007. The lower boundary for the July 2007 interval is July 1, 2007, regardless of whether the June 2007 partition was already created.

For interval partitioning, the partitioning key can only be a single column name from the table and it must be of `NUMBER` or `DATE` type. The optional `STORE IN` clause lets you specify one or more tablespaces into which the database will store interval partition data using a round-robin algorithm for subsequently created interval partitions.

The following example specifies four partitions with varying widths. It also specifies that above the transition point of January 1, 2007, partitions are created with a width of one month.

```
CREATE TABLE interval_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold    NUMBER(10,2)
    )
  PARTITION BY RANGE (time_id)
  INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))
    ( PARTITION p0 VALUES LESS THAN (TO_DATE('1-1-2005', 'DD-MM-YYYY')),
      PARTITION p1 VALUES LESS THAN (TO_DATE('1-1-2006', 'DD-MM-YYYY')),
      PARTITION p2 VALUES LESS THAN (TO_DATE('1-7-2006', 'DD-MM-YYYY')),
      PARTITION p3 VALUES LESS THAN (TO_DATE('1-1-2007', 'DD-MM-YYYY')) );
```

The high bound of partition `p3` represents the transition point. `p3` and all partitions below it (`p0`, `p1`, and `p2` in this example) are in the range section while all partitions above it fall into the interval section.

Creating Hash-Partitioned Tables and Global Indexes

The `PARTITION BY HASH` clause of the `CREATE TABLE` statement identifies that the table is to be hash-partitioned. The `PARTITIONS` clause can then be used to specify the number of partitions to create, and optionally, the tablespaces to store them in. Alternatively, you can use `PARTITION` clauses to name the individual partitions and their tablespaces.

The only attribute you can specify for hash partitions is `TABLESPACE`. All of the hash partitions of a table must share the same segment attributes (except `TABLESPACE`), which are inherited from the table level.

Creating a Hash Partitioned Table

The following example creates a hash-partitioned table. The partitioning column is `id`, four partitions are created and assigned system generated names, and they are placed in four named tablespaces (`gear1`, `gear2`, ...).

```
CREATE TABLE scubagear
     (id NUMBER,
      name VARCHAR2 (60))
   PARTITION BY HASH (id)
   PARTITIONS 4
   STORE IN (gear1, gear2, gear3, gear4);
```

See Also: "Using Multicolumn Partitioning Keys" on page 3-20

The following examples illustrate two methods of creating a hash-partitioned table named `dept`. In the first example the number of partitions is specified, but system generated names are assigned to them and they are stored in the default tablespace of the table.

```
CREATE TABLE dept (deptno NUMBER, deptname VARCHAR(32))
     PARTITION BY HASH(deptno) PARTITIONS 16;
```

In the following example, names of individual partitions, and tablespaces in which they are to reside, are specified. The initial extent size for each hash partition (segment) is also explicitly stated at the table level, and all partitions inherit this attribute.

```
CREATE TABLE dept (deptno NUMBER, deptname VARCHAR(32))
     STORAGE (INITIAL 10K)
     PARTITION BY HASH(deptno)
       (PARTITION p1 TABLESPACE ts1, PARTITION p2 TABLESPACE ts2,
        PARTITION p3 TABLESPACE ts1, PARTITION p4 TABLESPACE ts3);
```

If you create a local index for this table, the database constructs the index so that it is equipartitioned with the underlying table. The database also ensures that the index is maintained automatically when maintenance operations are performed on the underlying table. The following is an example of creating a local index on the table `dept`:

```
CREATE INDEX loc_dept_ix ON dept(deptno) LOCAL;
```

You can optionally name the hash partitions and tablespaces into which the local index partitions are to be stored, but if you do not do so, the database uses the name of the corresponding base partition as the index partition name, and stores the index partition in the same tablespace as the table partition.

Creating a Hash-Partitioned Global Index

Hash-partitioned global indexes can improve the performance of indexes where a small number of leaf blocks in the index have high contention in multiuser OLTP environments. Hash-partitioned global indexes can also limit the impact of index skew on monotonously increasing column values. Queries involving the equality and `IN` predicates on the index partitioning key can efficiently use hash-partitioned global indexes.

The syntax for creating a hash partitioned global index is similar to that used for a hash partitioned table. For example, the following statement creates a hash-partitioned global index:

```
CREATE INDEX hgidx ON tab (c1,c2,c3) GLOBAL
     PARTITION BY HASH (c1,c2)
     (PARTITION p1  TABLESPACE tbs_1,
      PARTITION p2  TABLESPACE tbs_2,
      PARTITION p3  TABLESPACE tbs_3,
      PARTITION p4  TABLESPACE tbs_4);
```

Creating List-Partitioned Tables

The semantics for creating list partitions are very similar to those for creating range partitions. However, to create list partitions, you specify a `PARTITION BY LIST` clause in the `CREATE TABLE` statement, and the `PARTITION` clauses specify lists of literal values, which are the discrete values of the partitioning columns that qualify rows to be included in the partition. For list partitioning, the partitioning key can only be a single column name from the table.

Available only with list partitioning, you can use the keyword `DEFAULT` to describe the value list for a partition. This identifies a partition that will accommodate rows that do not map into any of the other partitions.

As with range partitions, optional subclauses of a `PARTITION` clause can specify physical and other attributes specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their parent table.

The following example creates a list-partitioned table. It creates table `q1_sales_by_region` which is partitioned by regions consisting of groups of U.S. states.

```
CREATE TABLE q1_sales_by_region
      (deptno number,
       deptname varchar2(20),
       quarterly_sales number(10, 2),
       state varchar2(2))
   PARTITION BY LIST (state)
      (PARTITION q1_northwest VALUES ('OR', 'WA'),
       PARTITION q1_southwest VALUES ('AZ', 'UT', 'NM'),
       PARTITION q1_northeast VALUES  ('NY', 'VM', 'NJ'),
       PARTITION q1_southeast VALUES ('FL', 'GA'),
       PARTITION q1_northcentral VALUES ('SD', 'WI'),
       PARTITION q1_southcentral VALUES ('OK', 'TX'));
```

A row is mapped to a partition by checking whether the value of the partitioning column for a row matches a value in the value list that describes the partition.

For example, some sample rows are inserted as follows:

- (10, 'accounting', 100, 'WA') maps to partition `q1_northwest`
- (20, 'R&D', 150, 'OR') maps to partition `q1_northwest`
- (30, 'sales', 100, 'FL') maps to partition `q1_southeast`
- (40, 'HR', 10, 'TX') maps to partition `q1_southwest`
- (50, 'systems engineering', 10, 'CA') does not map to any partition in the table and raises an error

Unlike range partitioning, with list partitioning, there is no apparent sense of order between partitions. You can also specify a **default partition** into which rows that do not map to any other partition are mapped. If a default partition were specified in the preceding example, the state CA would map to that partition.

The following example creates table `sales_by_region` and partitions it using the list method. The first two `PARTITION` clauses specify physical attributes, which override the table-level defaults. The remaining `PARTITION` clauses do not specify attributes and those partitions inherit their physical attributes from table-level defaults. A default partition is also specified.

```
CREATE TABLE sales_by_region (item# INTEGER, qty INTEGER,
             store_name VARCHAR(30), state_code VARCHAR(2),
             sale_date DATE)
     STORAGE(INITIAL 10K NEXT 20K) TABLESPACE tbs5
     PARTITION BY LIST (state_code)
     (
     PARTITION region_east
        VALUES ('MA','NY','CT','NH','ME','MD','VA','PA','NJ')
        STORAGE (INITIAL 20K NEXT 40K PCTINCREASE 50)
        TABLESPACE tbs8,
     PARTITION region_west
        VALUES ('CA','AZ','NM','OR','WA','UT','NV','CO')
        NOLOGGING,
     PARTITION region_south
        VALUES ('TX','KY','TN','LA','MS','AR','AL','GA'),
     PARTITION region_central
        VALUES ('OH','ND','SD','MO','IL','MI','IA'),
     PARTITION region_null
        VALUES (NULL),
     PARTITION region_unknown
        VALUES (DEFAULT)
     );
```

Creating Reference-Partitioned Tables

To create a reference-partitioned table, you specify a `PARTITION BY REFERENCE` clause in the `CREATE TABLE` statement. This clause specifies the name of a referential constraint and this constraint becomes the partitioning referential constraint that is used as the basis for reference partitioning in the table. The referential constraint must be enabled and enforced.

As with other partitioned tables, you can specify object-level default attributes, and you can optionally specify partition descriptors that override the object-level defaults on a per-partition basis.

The following example creates a parent table `orders` which is range-partitioned on `order_date`. The reference-partitioned child table `order_items` is created with four partitions, `Q1_2005`, `Q2_2005`, `Q3_2005`, and `Q4_2005`, where each partition contains the `order_items` rows corresponding to orders in the respective parent partition.

```
CREATE TABLE orders
    ( order_id           NUMBER(12),
      order_date         TIMESTAMP WITH LOCAL TIME ZONE,
      order_mode         VARCHAR2(8),
      customer_id        NUMBER(6),
      order_status       NUMBER(2),
      order_total        NUMBER(8,2),
      sales_rep_id       NUMBER(6),
      promotion_id       NUMBER(6),
      CONSTRAINT orders_pk PRIMARY KEY(order_id)
    )
  PARTITION BY RANGE(order_date)
    ( PARTITION Q1_2005 VALUES LESS THAN (TO_DATE('01-APR-2005','DD-MON-YYYY')),
      PARTITION Q2_2005 VALUES LESS THAN (TO_DATE('01-JUL-2005','DD-MON-YYYY')),
      PARTITION Q3_2005 VALUES LESS THAN (TO_DATE('01-OCT-2005','DD-MON-YYYY')),
      PARTITION Q4_2005 VALUES LESS THAN (TO_DATE('01-JAN-2006','DD-MON-YYYY'))
    );

CREATE TABLE order_items
    ( order_id          NUMBER(12) NOT NULL,
      line_item_id      NUMBER(3)  NOT NULL,
      product_id        NUMBER(6)  NOT NULL,
      unit_price        NUMBER(8,2),
      quantity          NUMBER(8),
      CONSTRAINT order_items_fk
      FOREIGN KEY(order_id) REFERENCES orders(order_id)
    )
    PARTITION BY REFERENCE(order_items_fk);
```

If partition descriptors are provided, then the number of partitions described must exactly equal the number of partitions or subpartitions in the referenced table. If the parent table is a composite partitioned table, then the table will have one partition for each subpartition of its parent; otherwise the table will have one partition for each partition of its parent.

Partition bounds cannot be specified for the partitions of a reference-partitioned table.

The partitions of a reference-partitioned table can be named. If a partition is not explicitly named, then it will inherit its name from the corresponding partition in the parent table, unless this inherited name conflicts with one of the explicit names given. In this case, the partition will have a system-generated name.

Partitions of a reference-partitioned table will collocate with the corresponding partition of the parent table, if no explicit tablespace is specified for the reference-partitioned table's partition.

Creating Composite Partitioned Tables

To create a composite partitioned table, you start by using the `PARTITION BY [ RANGE | LIST ]` clause of a `CREATE TABLE` statement. Next, you specify a `SUBPARTITION BY [ RANGE | LIST | HASH ]` clause that follows similar syntax and rules as the `PARTITION BY [ RANGE | LIST | HASH ]` clause. The individual `PARTITION` and `SUBPARTITION` or `SUBPARTITIONS` clauses, and optionally a `SUBPARTITION TEMPLATE` clause, follow.

Creating Composite Range-Hash Partitioned Tables

The following statement creates a range-hash partitioned table. In this example, four range partitions are created, each containing eight subpartitions. Because the subpartitions are not named, system generated names are assigned, but the `STORE IN` clause distributes them across the 4 specified tablespaces (`ts1`, ...,`ts4`).

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) SUBPARTITION BY HASH (cust_id)
  SUBPARTITIONS 8 STORE IN (ts1, ts2, ts3, ts4)
 ( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
 , PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
 , PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
 , PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
 );
```

The partitions of a range-hash partitioned table are logical structures only, as their data is stored in the segments of their subpartitions. As with partitions, these subpartitions share the same logical attributes. Unlike range partitions in a range-partitioned table, the subpartitions cannot have different physical attributes from the owning partition, although they are not required to reside in the same tablespace.

Attributes specified for a range partition apply to all subpartitions of that partition. You can specify different attributes for each range partition, and you can specify a `STORE IN` clause at the partition level if the list of tablespaces across which the subpartitions of that partition should be spread is different from those of other partitions. All of this is illustrated in the following example.

```
CREATE TABLE emp (deptno NUMBER, empname VARCHAR(32), grade NUMBER)
     PARTITION BY RANGE(deptno) SUBPARTITION BY HASH(empname)
        SUBPARTITIONS 8 STORE IN (ts1, ts3, ts5, ts7)
    (PARTITION p1 VALUES LESS THAN (1000),
     PARTITION p2 VALUES LESS THAN (2000)
        STORE IN (ts2, ts4, ts6, ts8),
     PARTITION p3 VALUES LESS THAN (MAXVALUE)
       (SUBPARTITION p3_s1 TABLESPACE ts4,
        SUBPARTITION p3_s2 TABLESPACE ts5));
```

To learn how using a subpartition template can simplify the specification of a composite partitioned table, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

The following statement is an example of creating a local index on the `emp` table where the index segments are spread across tablespaces `ts7`, `ts8`, and `ts9`.

```
CREATE INDEX emp_ix ON emp(deptno)
     LOCAL STORE IN (ts7, ts8, ts9);
```

This local index is equipartitioned with the base table as follows:

- It consists of as many partitions as the base table.
- Each index partition consists of as many subpartitions as the corresponding base table partition.
- Index entries for rows in a given subpartition of the base table are stored in the corresponding subpartition of the index.

Creating Composite Range-List Partitioned Tables

The range partitions of a range-list composite partitioned table are described as for non-composite range partitioned tables. This allows that optional subclauses of a `PARTITION` clause can specify physical and other attributes, including tablespace, specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their underlying table.

The list subpartition descriptions, in the `SUBPARTITION` clauses, are described as for non-composite list partitions, except the only physical attribute that can be specified is a tablespace (optional). Subpartitions inherit all other physical attributes from the partition description.

The following example illustrates how range-list partitioning might be used. The example tracks sales data of products by quarters and within each quarter, groups it by specified states.

```
CREATE TABLE quarterly_regional_sales
      (deptno number, item_no varchar2(20),
       txn_date date, txn_amount number, state varchar2(2))
  TABLESPACE ts4
  PARTITION BY RANGE (txn_date)
    SUBPARTITION BY LIST (state)
      (PARTITION q1_1999 VALUES LESS THAN (TO_DATE('1-APR-1999','DD-MON-YYYY'))
         (SUBPARTITION q1_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q1_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q1_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q1_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q1_1999_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q1_1999_southcentral VALUES ('OK', 'TX')
         ),
       PARTITION q2_1999 VALUES LESS THAN ( TO_DATE('1-JUL-1999','DD-MON-YYYY'))
         (SUBPARTITION q2_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q2_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q2_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q2_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q2_1999_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q2_1999_southcentral VALUES ('OK', 'TX')
         ),
       PARTITION q3_1999 VALUES LESS THAN (TO_DATE('1-OCT-1999','DD-MON-YYYY'))
         (SUBPARTITION q3_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q3_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q3_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q3_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q3_1999_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q3_1999_southcentral VALUES ('OK', 'TX')
         ),
       PARTITION q4_1999 VALUES LESS THAN ( TO_DATE('1-JAN-2000','DD-MON-YYYY'))
         (SUBPARTITION q4_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q4_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q4_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q4_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q4_1999_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q4_1999_southcentral VALUES ('OK', 'TX')
         )
      );
```

A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within a specific partition range. The row is then mapped to a subpartition within that partition by identifying the subpartition whose descriptor value list contains a value matching the subpartition column value.

For example, some sample rows are inserted as follows:

- (10, 4532130, '23-Jan-1999', 8934.10, 'WA') maps to subpartition `q1_1999_northwest`
- (20, 5671621, '15-May-1999', 49021.21, 'OR') maps to subpartition `q2_1999_northwest`
- (30, 9977612, '07-Sep-1999', 30987.90, 'FL') maps to subpartition `q3_1999_southeast`

- (40, 9977612, '29-Nov-1999', 67891.45, 'TX') maps to subpartition `q4_1999_southcentral`
- (40, 4532130, '5-Jan-2000', 897231.55, 'TX') does not map to any partition in the table and raises an error
- (50, 5671621, '17-Dec-1999', 76123.35, 'CA') does not map to any subpartition in the table and raises an error

The partitions of a range-list partitioned table are logical structures only, as their data is stored in the segments of their subpartitions. The list subpartitions have the same characteristics as list partitions. You can specify a default subpartition, just as you specify a default partition for list partitioning.

The following example creates a table that specifies a tablespace at the partition and subpartition levels. The number of subpartitions within each partition varies, and default subpartitions are specified.

```
CREATE TABLE sample_regional_sales
      (deptno number, item_no varchar2(20),
       txn_date date, txn_amount number, state varchar2(2))
  PARTITION BY RANGE (txn_date)
    SUBPARTITION BY LIST (state)
      (PARTITION q1_1999 VALUES LESS THAN (TO_DATE('1-APR-1999','DD-MON-YYYY'))
          TABLESPACE tbs_1
         (SUBPARTITION q1_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q1_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q1_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q1_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q1_others VALUES (DEFAULT) TABLESPACE tbs_4
         ),
       PARTITION q2_1999 VALUES LESS THAN ( TO_DATE('1-JUL-1999','DD-MON-YYYY'))
          TABLESPACE tbs_2
         (SUBPARTITION q2_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q2_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q2_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q2_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q2_1999_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q2_1999_southcentral VALUES ('OK', 'TX')
         ),
       PARTITION q3_1999 VALUES LESS THAN (TO_DATE('1-OCT-1999','DD-MON-YYYY'))
          TABLESPACE tbs_3
         (SUBPARTITION q3_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q3_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q3_others VALUES (DEFAULT) TABLESPACE tbs_4
         ),
       PARTITION q4_1999 VALUES LESS THAN ( TO_DATE('1-JAN-2000','DD-MON-YYYY'))
          TABLESPACE tbs_4
      );
```

This example results in the following subpartition descriptions:

- All subpartitions inherit their physical attributes, other than tablespace, from tablespace level defaults. This is because the only physical attribute that has been specified for partitions or subpartitions is tablespace. There are no table level physical attributes specified, thus tablespace level defaults are inherited at all levels.
- The first 4 subpartitions of partition `q1_1999` are all contained in `tbs_1`, except for the subpartition `q1_others`, which is stored in `tbs_4` and contains all rows that do not map to any of the other partitions.

- The 6 subpartitions of partition `q2_1999` are all stored in `tbs_2`.
- The first 2 subpartitions of partition `q3_1999` are all contained in `tbs_3`, except for the subpartition `q3_others`, which is stored in `tbs_4` and contains all rows that do not map to any of the other partitions.
- There is no subpartition description for partition `q4_1999`. This results in one default subpartition being created and stored in `tbs_4`. The subpartition name is system generated in the form `SYS_SUBP`*n*.

To learn how using a subpartition template can simplify the specification of a composite partitioned table, see "Using Subpartition Templates to Describe Composite Partitioned Tables".

Creating Composite Range-Range Partitioned Tables

The range partitions of a range-range composite partitioned table are described as for non-composite range partitioned tables. This allows that optional subclauses of a `PARTITION` clause can specify physical and other attributes, including tablespace, specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their underlying table.

The range subpartition descriptions, in the `SUBPARTITION` clauses, are described as for non-composite range partitions, except the only physical attribute that can be specified is an optional tablespace. Subpartitions inherit all other physical attributes from the partition description.

The following example illustrates how range-range partitioning might be used. The example tracks shipments. The service level agreement with the customer states that every order will be delivered in the calendar month after the order was placed. The following types of orders are identified:

- E (EARLY): orders that are delivered before the the middle of the next month after the order was placed. These orders likely exceed customers' expectations.
- A (AGREED): orders that are delivered in the calendar month after the order was placed (but not early orders).
- L (LATE): orders that were only delivered starting the second calendar month after the order was placed.

```
CREATE TABLE shipments
( order_id      NUMBER NOT NULL
, order_date    DATE NOT NULL
, delivery_date DATE NOT NULL
, customer_id   NUMBER NOT NULL
, sales_amount  NUMBER NOT NULL
)
PARTITION BY RANGE (order_date)
SUBPARTITION BY RANGE (delivery_date)
( PARTITION p_2006_jul VALUES LESS THAN (TO_DATE('01-AUG-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_jul_e VALUES LESS THAN (TO_DATE('15-AUG-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_jul_a VALUES LESS THAN (TO_DATE('01-SEP-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_jul_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_aug VALUES LESS THAN (TO_DATE('01-SEP-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_aug_e VALUES LESS THAN (TO_DATE('15-SEP-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_aug_a VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_aug_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_sep VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_sep_e VALUES LESS THAN (TO_DATE('15-OCT-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_sep_a VALUES LESS THAN (TO_DATE('01-NOV-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_sep_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_oct VALUES LESS THAN (TO_DATE('01-NOV-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_oct_e VALUES LESS THAN (TO_DATE('15-NOV-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_oct_a VALUES LESS THAN (TO_DATE('01-DEC-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_oct_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_nov VALUES LESS THAN (TO_DATE('01-DEC-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_nov_e VALUES LESS THAN (TO_DATE('15-DEC-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_nov_a VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
  , SUBPARTITION p06_nov_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_dec VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
  ( SUBPARTITION p06_dec_e VALUES LESS THAN (TO_DATE('15-JAN-2007','dd-MON-yyyy'))
  , SUBPARTITION p06_dec_a VALUES LESS THAN (TO_DATE('01-FEB-2007','dd-MON-yyyy'))
  , SUBPARTITION p06_dec_l VALUES LESS THAN (MAXVALUE)
  )
);
```

A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within a specific partition range. The row is then mapped to a subpartition within that partition by identifying whether the value of the subpartitioning column falls within a specific range. For example, a shipment with an order date in September 2006 and a delivery date of October 28, 2006 falls in partition `p06_oct_a`.

To learn how using a subpartition template can simplify the specification of a composite partitioned table, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

Creating Composite List-* Partitioned Tables

The concepts of list-hash, list-list, and list-range composite partitioning are similar to the concepts for range-hash, range-list, and range-range partitioning. This time, however, you specify `PARTITION BY LIST` to define the partitioning strategy.

The list partitions of a list-* composite partitioned table are described as for non-composite range partitioned tables. This allows that optional subclauses of a `PARTITION` clause can specify physical and other attributes, including tablespace, specific to a partition segment. If not overridden at the partition level, then partitions inherit the attributes of their underlying table.

The subpartition descriptions, in the `SUBPARTITION` or `SUBPARTITIONS` clauses, are described as for range-* composite partitioning methods.

See Also:

- "Creating Composite Range-Hash Partitioned Tables" on page 3-8 for more details on the subpartition definition of a list-hash composite partitioning method
- "Creating Composite Range-List Partitioned Tables" on page 3-9 for more details on the subpartition definition of a list-list composite partitioning method
- "Creating Composite Range-Range Partitioned Tables" on page 3-12 for more details on the subpartition definition of a list-range composite partitioning method

The following sections show examples for the different list-* composite partitioning methods.

Creating Composite List-Hash Partitioned Tables The following example shows an `accounts` table that is list partitioned by region and subpartitioned using hash by customer identifier.

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, balance        NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY HASH (customer_id) SUBPARTITIONS 8
( PARTITION p_northwest VALUES ('OR', 'WA')
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
, PARTITION p_southeast VALUES ('FL', 'GA')
, PARTITION p_northcentral VALUES ('SD', 'WI')
, PARTITION p_southcentral VALUES ('OK', 'TX')
);
```

To learn how using a subpartition template can simplify the specification of a composite partitioned table, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

Creating Composite List-List Partitioned Tables The following example shows an `accounts` table that is list partitioned by region and subpartitioned using list by account status.

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, balance        NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY LIST (status)
( PARTITION p_northwest VALUES ('OR', 'WA')
  ( SUBPARTITION p_nw_bad VALUES ('B')
  , SUBPARTITION p_nw_average VALUES ('A')
  , SUBPARTITION p_nw_good VALUES ('G')
  )
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
  ( SUBPARTITION p_sw_bad VALUES ('B')
  , SUBPARTITION p_sw_average VALUES ('A')
  , SUBPARTITION p_sw_good VALUES ('G')
  )
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
  ( SUBPARTITION p_ne_bad VALUES ('B')
  , SUBPARTITION p_ne_average VALUES ('A')
  , SUBPARTITION p_ne_good VALUES ('G')
  )
, PARTITION p_southeast VALUES ('FL', 'GA')
  ( SUBPARTITION p_se_bad VALUES ('B')
  , SUBPARTITION p_se_average VALUES ('A')
  , SUBPARTITION p_se_good VALUES ('G')
  )
, PARTITION p_northcentral VALUES ('SD', 'WI')
  ( SUBPARTITION p_nc_bad VALUES ('B')
  , SUBPARTITION p_nc_average VALUES ('A')
  , SUBPARTITION p_nc_good VALUES ('G')
  )
, PARTITION p_southcentral VALUES ('OK', 'TX')
  ( SUBPARTITION p_sc_bad VALUES ('B')
  , SUBPARTITION p_sc_average VALUES ('A')
  , SUBPARTITION p_sc_good VALUES ('G')
  )
);
```

To learn how using a subpartition template can simplify the specification of a composite partitioned table, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

Creating Composite List-Range Partitioned Tables The following example shows an `accounts` table that is list partitioned by region and subpartitioned using range by account balance. Note that row movement is enabled. Subpartitions for different list partitions could have different ranges specified.

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, balance        NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY RANGE (balance)
( PARTITION p_northwest VALUES ('OR', 'WA')
  ( SUBPARTITION p_nw_low VALUES LESS THAN (1000)
  , SUBPARTITION p_nw_average VALUES LESS THAN (10000)
  , SUBPARTITION p_nw_high VALUES LESS THAN (100000)
  , SUBPARTITION p_nw_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
  ( SUBPARTITION p_sw_low VALUES LESS THAN (1000)
  , SUBPARTITION p_sw_average VALUES LESS THAN (10000)
  , SUBPARTITION p_sw_high VALUES LESS THAN (100000)
  , SUBPARTITION p_sw_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
  ( SUBPARTITION p_ne_low VALUES LESS THAN (1000)
  , SUBPARTITION p_ne_average VALUES LESS THAN (10000)
  , SUBPARTITION p_ne_high VALUES LESS THAN (100000)
  , SUBPARTITION p_ne_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_southeast VALUES ('FL', 'GA')
  ( SUBPARTITION p_se_low VALUES LESS THAN (1000)
  , SUBPARTITION p_se_average VALUES LESS THAN (10000)
  , SUBPARTITION p_se_high VALUES LESS THAN (100000)
  , SUBPARTITION p_se_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_northcentral VALUES ('SD', 'WI')
  ( SUBPARTITION p_nc_low VALUES LESS THAN (1000)
  , SUBPARTITION p_nc_average VALUES LESS THAN (10000)
  , SUBPARTITION p_nc_high VALUES LESS THAN (100000)
  , SUBPARTITION p_nc_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_southcentral VALUES ('OK', 'TX')
  ( SUBPARTITION p_sc_low VALUES LESS THAN (1000)
  , SUBPARTITION p_sc_average VALUES LESS THAN (10000)
  , SUBPARTITION p_sc_high VALUES LESS THAN (100000)
  , SUBPARTITION p_sc_extraordinary VALUES LESS THAN (MAXVALUE)
  )
) ENABLE ROW MOVEMENT;
```

To learn how using a subpartition template can simplify the specification of a composite partitioned table, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

Creating Composite Interval-* Partitioned Tables

The concepts of interval-* composite partitioning are similar to the concepts for range-* partitioning. However, you extend the `PARTITION BY RANGE` clause to include the `INTERVAL` definition. You must specify at least one range partition using the `PARTITION` clause. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database automatically creates interval partitions for data beyond that transition point.

The subpartitions for intervals in an interval-* partitioned table will be created when the database creates the interval. You can specify the definition of future subpartitions only through the use of a subpartition template. To learn more about how to use a subpartition template, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

Creating Composite Interval-Hash Partitioned Tables You can create an interval-hash partitioned table with multiple hash partitions using one of the following methods:

- Specify a number of hash partitions in the `PARTITIONS` clause.
- Use a subpartition template.

If you do not use either of these methods, then future interval partitions will only get a single hash subpartition.

The following example shows the `sales` table, interval partitioned using monthly intervals on `time_id`, with hash subpartitions by `cust_id`. Note that this example specifies a number of hash partitions, without any specific tablespace assignment to the individual hash partitions.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY HASH (cust_id) SUBPARTITIONS 4
 ( PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy')))
```

```
PARALLEL;
```

The following example shows the same `sales` table, interval partitioned using monthly intervals on `time_id`, again with hash subpartitions by `cust_id`. This time, however, individual hash partitions will be stored in separate tablespaces. Note that the subpartition template is used in order to define the tablespace assignment for future hash subpartitions. To learn more about how to use a subpartition template, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY hash(cust_id)
   SUBPARTITION template
   ( SUBPARTITION p1 TABLESPACE ts1
   , SUBPARTITION p2 TABLESPACE ts2
   , SUBPARTITION p3 TABLESPACE ts3
   , SUBPARTITION P4 TABLESPACE ts4
   )
 ( PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy'))
) PARALLEL;
```

Creating Composite Interval-List Partitioned Tables The only way to define list subpartitions for future interval partitions is through the use of the subpartition template. If you do not use the subpartitioning template, then the only subpartition that will be created for every interval partition is a `DEFAULT` subpartition. To learn more about how to use a subpartition template, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

The following example shows the `sales` table, interval partitioned using daily intervals on `time_id`, with list subpartitions by `channel_id`.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY RANGE(amount_sold)
   SUBPARTITION TEMPLATE
   ( SUBPARTITION p_low VALUES LESS THAN (1000)
   , SUBPARTITION p_medium VALUES LESS THAN (4000)
   , SUBPARTITION p_high VALUES LESS THAN (8000)
   , SUBPARTITION p_ultimate VALUES LESS THAN (maxvalue)
   )
 ( PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy')))
PARALLEL;
```

Creating Composite Interval-Range Partitioned Tables The only way to define range subpartitions for future interval partitions is through the use of the subpartition template. If you do not use the subpartition template, then the only subpartition that will be created for every interval partition is a range subpartition with the `MAXVALUE` upper boundary. To learn more about how to use a subpartition template, see "Using Subpartition Templates to Describe Composite Partitioned Tables" on page 3-18.

The following example shows the `sales` table, interval partitioned using daily intervals on `time_id`, with range subpartitions by `amount_sold`.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTODSINTERVAL(1,'DAY'))
 SUBPARTITION BY LIST (channel_id)
   SUBPARTITION TEMPLATE
   ( SUBPARTITION p_catalog VALUES ('C')
   , SUBPARTITION p_internet VALUES ('I')
   , SUBPARTITION p_partners VALUES ('P')
   , SUBPARTITION p_direct_sales VALUES ('S')
   , SUBPARTITION p_tele_sales VALUES ('T')
   )
 ( PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy')))
PARALLEL;
```

Using Subpartition Templates to Describe Composite Partitioned Tables

You can create subpartitions in a composite partitioned table using a subpartition template. A subpartition template simplifies the specification of subpartitions by not requiring that a subpartition descriptor be specified for every partition in the table. Instead, you describe subpartitions only once in a template, then apply that subpartition template to every partition in the table. For interval-* composite partitioned tables, the subpartition template is the only way to define subpartitions for interval partitions.

The subpartition template is used whenever a subpartition descriptor is not specified for a partition. If a subpartition descriptor is specified, then it is used instead of the subpartition template for that partition. If no subpartition template is specified, and no subpartition descriptor is supplied for a partition, then a single default subpartition is created.

Specifying a Subpartition Template for a *-Hash Partitioned Table

In the case of [range | interval | list]-hash partitioned tables, the subpartition template can describe the subpartitions in detail, or it can specify just the number of hash subpartitions.

The following example creates a range-hash partitioned table using a subpartition template:

```
CREATE TABLE emp_sub_template (deptno NUMBER, empname VARCHAR(32), grade NUMBER)
```

```
     PARTITION BY RANGE(deptno) SUBPARTITION BY HASH(empname)
     SUBPARTITION TEMPLATE
         (SUBPARTITION a TABLESPACE ts1,
          SUBPARTITION b TABLESPACE ts2,
          SUBPARTITION c TABLESPACE ts3,
          SUBPARTITION d TABLESPACE ts4
         )
    (PARTITION p1 VALUES LESS THAN (1000),
     PARTITION p2 VALUES LESS THAN (2000),
     PARTITION p3 VALUES LESS THAN (MAXVALUE)
    );
```

This example produces the following table description:

- Every partition has four subpartitions as described in the subpartition template.
- Each subpartition has a tablespace specified. It is required that if a tablespace is specified for one subpartition in a subpartition template, then one must be specified for all.
- The names of the subpartitions, unless you use interval-* subpartitioning, are generated by concatenating the partition name with the subpartition name in the form:

  partition name_subpartition name

  For interval-* subpartitioning, the subpartition names are system-generated in the form:

  `SYS_SUBP`*n*

The following query displays the subpartition names and tablespaces:

```
SQL> SELECT TABLESPACE_NAME, PARTITION_NAME, SUBPARTITION_NAME
  2  FROM DBA_TAB_SUBPARTITIONS WHERE TABLE_NAME='EMP_SUB_TEMPLATE'
  3  ORDER BY TABLESPACE_NAME;

TABLESPACE_NAME PARTITION_NAME  SUBPARTITION_NAME
--------------- --------------- ------------------
TS1             P1              P1_A
TS1             P2              P2_A
TS1             P3              P3_A
TS2             P1              P1_B
TS2             P2              P2_B
TS2             P3              P3_B
TS3             P1              P1_C
TS3             P2              P2_C
TS3             P3              P3_C
TS4             P1              P1_D
TS4             P2              P2_D
TS4             P3              P3_D

12 rows selected.
```

Specifying a Subpartition Template for a *-List Partitioned Table

The following example, for a range-list partitioned table, illustrates how using a subpartition template can help you stripe data across tablespaces. In this example a table is created where the table subpartitions are vertically striped, meaning that subpartition *n* from every partition is in the same tablespace.

```
CREATE TABLE stripe_regional_sales
            ( deptno number, item_no varchar2(20),
              txn_date date, txn_amount number, state varchar2(2))
   PARTITION BY RANGE (txn_date)
   SUBPARTITION BY LIST (state)
   SUBPARTITION TEMPLATE
      (SUBPARTITION northwest VALUES ('OR', 'WA') TABLESPACE tbs_1,
       SUBPARTITION southwest VALUES ('AZ', 'UT', 'NM') TABLESPACE tbs_2,
       SUBPARTITION northeast VALUES ('NY', 'VM', 'NJ') TABLESPACE tbs_3,
       SUBPARTITION southeast VALUES ('FL', 'GA') TABLESPACE tbs_4,
       SUBPARTITION midwest VALUES ('SD', 'WI') TABLESPACE tbs_5,
       SUBPARTITION south VALUES ('AL', 'AK') TABLESPACE tbs_6,
       SUBPARTITION others VALUES (DEFAULT ) TABLESPACE tbs_7
      )
  (PARTITION q1_1999 VALUES LESS THAN ( TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION q2_1999 VALUES LESS THAN ( TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION q3_1999 VALUES LESS THAN ( TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION q4_1999 VALUES LESS THAN ( TO_DATE('1-JAN-2000','DD-MON-YYYY'))
  );
```

If you specified the tablespaces at the partition level (for example, `tbs_1` for partition `q1_1999`, `tbs_2` for partition `q2_1999`, `tbs_3` for partition `q3_1999`, and `tbs_4` for partition `q4_1999`) and not in the subpartition template, then the table would be horizontally striped. All subpartitions would be in the tablespace of the owning partition.

Using Multicolumn Partitioning Keys

For range-partitioned and hash-partitioned tables, you can specify up to 16 partitioning key columns. Multicolumn partitioning should be used when the partitioning key is composed of several columns and subsequent columns define a higher granularity than the preceding ones. The most common scenario is a decomposed `DATE` or `TIMESTAMP` key, consisting of separated columns, for year, month, and day.

In evaluating multicolumn partitioning keys, the database uses the second value only if the first value cannot uniquely identify a single target partition, and uses the third value only if the first and second do not determine the correct partition, and so forth. A value cannot determine the correct partition only when a partition bound exactly matches that value and the same bound is defined for the next partition. The n^{th} column will therefore be investigated only when all previous (n-1) values of the multicolumn key exactly match the (n-1) bounds of a partition. A second column, for example, will be evaluated only if the first column exactly matches the partition boundary value. If all column values exactly match all of the bound values for a partition, then the database will determine that the row does not fit in this partition and will consider the next partition for a match.

In the case of nondeterministic boundary definitions (successive partitions with identical values for at least one column), the partition boundary value becomes an inclusive value, representing a "less than or equal to" boundary. This is in contrast to deterministic boundaries, where the values are always regarded as "less than" boundaries.

The following example illustrates the column evaluation for a multicolumn range-partitioned table, storing the actual `DATE` information in three separate columns: `year`, `month`, and `day`. The partitioning granularity is a calendar quarter. The partitioned table being evaluated is created as follows:

```
CREATE TABLE sales_demo (
   year          NUMBER,
   month         NUMBER,
   day           NUMBER,
   amount_sold   NUMBER)
PARTITION BY RANGE (year,month)
  (PARTITION before2001 VALUES LESS THAN (2001,1),
   PARTITION q1_2001    VALUES LESS THAN (2001,4),
   PARTITION q2_2001    VALUES LESS THAN (2001,7),
   PARTITION q3_2001    VALUES LESS THAN (2001,10),
   PARTITION q4_2001    VALUES LESS THAN (2002,1),
   PARTITION future     VALUES LESS THAN (MAXVALUE,0));

REM  12-DEC-2000
INSERT INTO sales_demo VALUES(2000,12,12, 1000);
REM  17-MAR-2001
INSERT INTO sales_demo VALUES(2001,3,17, 2000);
REM  1-NOV-2001
INSERT INTO sales_demo VALUES(2001,11,1, 5000);
REM  1-JAN-2002
INSERT INTO sales_demo VALUES(2002,1,1, 4000);
```

The year value for 12-DEC-2000 satisfied the first partition, `before2001`, so no further evaluation is needed:

```
SELECT * FROM sales_demo PARTITION(before2001);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2000         12         12        1000
```

The information for 17-MAR-2001 is stored in partition `q1_2001`. The first partitioning key column, `year`, does not by itself determine the correct partition, so the second partitioning key column, `month`, must be evaluated.

```
SELECT * FROM sales_demo PARTITION(q1_2001);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2001          3         17        2000
```

Following the same determination rule as for the previous record, the second column, `month`, determines partition `q4_2001` as correct partition for 1-NOV-2001:

```
SELECT * FROM sales_demo PARTITION(q4_2001);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2001         11          1        5000
```

The partition for 01-JAN-2002 is determined by evaluating only the `year` column, which indicates the `future` partition:

```
SELECT * FROM sales_demo PARTITION(future);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2002          1          1        4000
```

If the database encounters `MAXVALUE` in one of the partitioning key columns, then all other values of subsequent columns become irrelevant. That is, a definition of partition `future` in the preceding example, having a bound of (`MAXVALUE`,0) is equivalent to a bound of (`MAXVALUE`,100) or a bound of (`MAXVALUE`,`MAXVALUE`).

The following example illustrates the use of a multicolumn partitioned approach for table `supplier_parts`, storing the information about which suppliers deliver which parts. To distribute the data in equal-sized partitions, it is not sufficient to partition the table based on the `supplier_id`, because some suppliers might provide hundreds of thousands of parts, while others provide only a few specialty parts. Instead, you partition the table on (`supplier_id`, `partnum`) to manually enforce equal-sized partitions.

```
CREATE TABLE supplier_parts (
   supplier_id      NUMBER,
   partnum          NUMBER,
   price            NUMBER)
PARTITION BY RANGE (supplier_id, partnum)
  (PARTITION p1 VALUES LESS THAN  (10,100),
   PARTITION p2 VALUES LESS THAN (10,200),
   PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE));
```

The following three records are inserted into the table:

```
INSERT INTO supplier_parts VALUES (5,5, 1000);
INSERT INTO supplier_parts VALUES (5,150, 1000);
INSERT INTO supplier_parts VALUES (10,100, 1000);
```

The first two records are inserted into partition `p1`, uniquely identified by `supplier_id`. However, the third record is inserted into partition `p2`; it matches all range boundary values of partition `p1` exactly and the database therefore considers the following partition for a match. The value of `partnum` satisfies the criteria < 200, so it is inserted into partition `p2`.

```
SELECT * FROM supplier_parts PARTITION (p1);

SUPPLIER_ID    PARTNUM      PRICE
----------- ---------- ----------
          5          5       1000
          5        150       1000

SELECT * FROM supplier_parts PARTITION (p2);

SUPPLIER_ID    PARTNUM      PRICE
----------- ---------- ----------
         10        100       1000
```

Every row with `supplier_id` < 10 will be stored in partition `p1`, regardless of the `partnum` value. The column `partnum` will be evaluated only if `supplier_id` =10, and the corresponding rows will be inserted into partition `p1`, `p2`, or even into `p3` when `partnum` >=200. To achieve equal-sized partitions for ranges of `supplier_parts`, you could choose a composite range-hash partitioned table, range partitioned by `supplier_id`, hash subpartitioned by `partnum`.

Defining the partition boundaries for multicolumn partitioned tables must obey some rules. For example, consider a table that is range partitioned on three columns `a`, `b`, and `c`. The individual partitions have range values represented as follows:

```
P0(a0, b0, c0)
P1(a1, b1, c1)
P2(a2, b2, c2)
...
Pn(an, bn, cn)
```

The range values you provide for each partition must follow these rules:

- `a0` must be less than or equal to `a1`, and `a1` must be less than or equal to `a2`, and so on.
- If `a0=a1`, then `b0` must be less than or equal to `b1`. If `a0 < a1`, then `b0` and `b1` can have any values. If `a0=a1` and `b0=b1`, then `c0` must be less than or equal to `c1`. If `b0<b1`, then `c0` and `c1` can have any values, and so on.
- If `a1=a2`, then `b1` must be less than or equal to `b2`. If `a1<a2`, then `b1` and `b2` can have any values. If `a1=a2` and `b1=b2`, then `c1` must be less than or equal to `c2`. If `b1<b2`, then `c1` and `c2` can have any values, and so on.

Using Virtual Column-Based Partitioning

In the context of partitioning, a virtual column can be used as any regular column. All partition methods are supported when using virtual columns, including interval partitioning and all different combinations of composite partitioning. A virtual column that you want to use as the partitioning column cannot use calls to a PL/SQL function.

> **See Also:** *Oracle Database SQL Language Reference* for the syntax on how to create a virtual column

The following example shows the `sales` table partitioned by range-range using a virtual column for the subpartitioning key. The virtual column calculates the total value of a sale by multiplying `amount_sold` and `quantity_sold`.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6) NOT NULL
  , cust_id       NUMBER NOT NULL
  , time_id       DATE NOT NULL
  , channel_id    CHAR(1) NOT NULL
  , promo_id      NUMBER(6) NOT NULL
  , quantity_sold NUMBER(3) NOT NULL
  , amount_sold   NUMBER(10,2) NOT NULL
  , total_amount AS (quantity_sold * amount_sold)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY RANGE(total_amount)
 SUBPARTITION TEMPLATE
   ( SUBPARTITION p_small VALUES LESS THAN (1000)
   , SUBPARTITION p_medium VALUES LESS THAN (5000)
   , SUBPARTITION p_large VALUES LESS THAN (10000)
   , SUBPARTITION p_extreme VALUES LESS THAN (MAXVALUE)
   )
 (PARTITION sales_before_2007 VALUES LESS THAN
        (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
)
ENABLE ROW MOVEMENT
PARALLEL NOLOGGING;
```

As the example shows, row movement is also supported with virtual columns. If row movement is enabled, then a row will migrate from one partition to another partition if the virtual column evaluates to a value that belongs to another partition.

Using Table Compression with Partitioned Tables

For heap-organized partitioned tables, you can compress some or all partitions using table compression. The compression attribute can be declared for a tablespace, a table, or a partition of a table. Whenever the compress attribute is not specified, it is inherited like any other storage attribute.

The following example creates a list-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));
```

Using Key Compression with Partitioned Indexes

You can compress some or all partitions of a B-tree index using key compression. Key compression is applicable only to B-tree indexes. Bitmap indexes are stored in a compressed manner by default. An index using key compression eliminates repeated occurrences of key column prefix values, thus saving space and I/O.

The following example creates a local partitioned index with all partitions except the most recent one compressed:

```
CREATE INDEX i_cost1 ON costs_demo (prod_id) COMPRESS LOCAL
   (PARTITION costs_old, PARTITION costs_q1_2003,
    PARTITION costs_q2_2003, PARTITION costs_recent NOCOMPRESS);
```

You cannot specify `COMPRESS` (or `NOCOMPRESS`) explicitly for an index subpartition. All index subpartitions of a given partition inherit the key compression setting from the parent partition.

To modify the key compression attribute for all subpartitions of a given partition, you must first issue an `ALTER INDEX...MODIFY PARTITION` statement and then rebuild all subpartitions. The `MODIFY PARTITION` clause will mark all index subpartitions as `UNUSABLE`.

Creating Partitioned Index-Organized Tables

For index-organized tables, you can use the range, list, or hash partitioning method. The semantics for creating partitioned index-organized tables is similar to that for regular tables with these differences:

- When you create the table, you specify the `ORGANIZATION INDEX` clause, and `INCLUDING` and `OVERFLOW` clauses as necessary.
- The `PARTITION` or `PARTITIONS` clauses can have `OVERFLOW` subclauses that allow you to specify attributes of the overflow segments at the partition level.

Specifying an `OVERFLOW` clause results in the overflow data segments themselves being equipartitioned with the primary key index segments. Thus, for partitioned index-organized tables with overflow, each partition has an index segment and an overflow data segment.

For index-organized tables, the set of partitioning columns must be a subset of the primary key columns. Because rows of an index-organized table are stored in the primary key index for the table, the partitioning criterion has an effect on the availability. By choosing the partitioning key to be a subset of the primary key, an insert operation only needs to verify uniqueness of the primary key in a single partition, thereby maintaining partition independence.

Support for secondary indexes on index-organized tables is similar to the support for regular tables. Because of the logical nature of the secondary indexes, global indexes on index-organized tables remain usable for certain operations where they would be marked UNUSABLE for regular tables.

See Also:

- *Oracle Database Administrator's Guide* for more information about managing index-organized tables
- "Maintaining Partitions" on page 3-27
- *Oracle Database Concepts* for more information about index-organized tables

Creating Range-Partitioned Index-Organized Tables

You can partition index-organized tables, and their secondary indexes, by the range method. In the following example, a range-partitioned index-organized table `sales` is created. The `INCLUDING` clause specifies that all columns after `week_no` are to be stored in an overflow segment. There is one overflow segment for each partition, all stored in the same tablespace (`overflow_here`). Optionally, `OVERFLOW TABLESPACE` could be specified at the individual partition level, in which case some or all of the overflow segments could have separate `TABLESPACE` attributes.

```
CREATE TABLE sales(acct_no NUMBER(5),
                   acct_name CHAR(30),
                   amount_of_sale NUMBER(6),
                   week_no INTEGER,
                   sale_details VARCHAR2(1000),
             PRIMARY KEY (acct_no, acct_name, week_no))
     ORGANIZATION INDEX
             INCLUDING week_no
             OVERFLOW TABLESPACE overflow_here
     PARTITION BY RANGE (week_no)
            (PARTITION VALUES LESS THAN (5)
                   TABLESPACE ts1,
             PARTITION VALUES LESS THAN (9)
                   TABLESPACE ts2 OVERFLOW TABLESPACE overflow_ts2,
             ...
             PARTITION VALUES LESS THAN (MAXVALUE)
                   TABLESPACE ts13);
```

Creating Hash-Partitioned Index-Organized Tables

Another option for partitioning index-organized tables is to use the hash method. In the following example, the `sales` index-organized table is partitioned by the hash method.

```
CREATE TABLE sales(acct_no NUMBER(5),
                   acct_name CHAR(30),
                   amount_of_sale NUMBER(6),
                   week_no INTEGER,
                   sale_details VARCHAR2(1000),
             PRIMARY KEY (acct_no, acct_name, week_no))
     ORGANIZATION INDEX
             INCLUDING week_no
     OVERFLOW
          PARTITION BY HASH (week_no)
```

```
PARTITIONS 16
STORE IN (ts1, ts2, ts3, ts4)
OVERFLOW STORE IN (ts3, ts6, ts9);
```

Note: A well-designed hash function is intended to distribute rows in a well-balanced fashion among the partitions. Therefore, updating the primary key column(s) of a row is very likely to move that row to a different partition. Oracle recommends that you explicitly specify the `ENABLE ROW MOVEMENT` clause when creating a hash-partitioned index-organized table with a changeable partitioning key. The default is that `ENABLE ROW MOVEMENT` is disabled.

Creating List-Partitioned Index-Organized Tables

The other option for partitioning index-organized tables is to use the list method. In the following example, the `sales` index-organized table is partitioned by the list method. This example uses the `example` tablespace, which is part of the sample schemas in your seed database. Normally you would specify different tablespace storage for different partitions.

```
CREATE TABLE sales(acct_no NUMBER(5),
                   acct_name CHAR(30),
                   amount_of_sale NUMBER(6),
                   week_no INTEGER,
                   sale_details VARCHAR2(1000),
             PRIMARY KEY (acct_no, acct_name, week_no))
     ORGANIZATION INDEX
             INCLUDING week_no
             OVERFLOW TABLESPACE example
     PARTITION BY LIST (week_no)
            (PARTITION VALUES (1, 2, 3, 4)
                   TABLESPACE example,
             PARTITION VALUES (5, 6, 7, 8)
                   TABLESPACE example OVERFLOW TABLESPACE example,
             PARTITION VALUES (DEFAULT)
                   TABLESPACE example);
```

Partitioning Restrictions for Multiple Block Sizes

Use caution when creating partitioned objects in a database with tablespaces of different block sizes. The storage of partitioned objects in such tablespaces is subject to some restrictions. Specifically, all partitions of the following entities must reside in tablespaces of the same block size:

- Conventional tables
- Indexes
- Primary key index segments of index-organized tables
- Overflow segments of index-organized tables
- `LOB` columns stored out of line

Therefore:

- For each conventional table, all partitions of that table must be stored in tablespaces with the same block size.
- For each index-organized table, all primary key index partitions must reside in tablespaces of the same block size, and all overflow partitions of that table must

reside in tablespaces of the same block size. However, index partitions and overflow partitions can reside in tablespaces of different block size.

- For each index (global or local), each partition of that index must reside in tablespaces of the same block size. However, partitions of different indexes defined on the same object can reside in tablespaces of different block sizes.
- For each `LOB` column, each partition of that column must be stored in tablespaces of equal block sizes. However, different `LOB` columns can be stored in tablespaces of different block sizes.

When you create or alter a partitioned table or index, all tablespaces you *explicitly specify* for the partitions and subpartitions of each entity must be of the same block size. If you *do not explicitly specify* tablespace storage for an entity, the tablespaces the database uses by default must be of the same block size. Therefore you must be aware of the default tablespaces at each level of the partitioned object.

Maintaining Partitions

This section describes how to perform partition and subpartition maintenance operations for both tables and indexes.

Table 3–1 lists partition maintenance operations that can be performed on partitioned tables and composite partitioned tables, and Table 3–2 lists subpartition maintenance operations that can be performed on composite partitioned tables. For each type of partitioning and subpartitioning, the specific clause of the `ALTER TABLE` statement that is used to perform that maintenance operation is listed.

Table 3–1 ALTER TABLE Maintenance Operations for Table Partitions

Maintenance Operation	**Range** **Composite Range-***	**Interval** **Composite Interval-***	**Hash**	**List** **Composite List-***	**Reference**
Adding Partitions	`ADD PARTITION`	`ADD PARTITION`	`ADD PARTITION`	`ADD PARTITION`	N/A[1]
Coalescing Partitions	N/A	N/A	`COALESCE PARTITION`	N/A	N/A[1]
Dropping Partitions	`DROP PARTITION`	`DROP PARTITION`	N/A	`DROP PARTITION`	N/A[1]
Exchanging Partitions	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`
Merging Partitions	`MERGE PARTITIONS`	`MERGE PARTITIONS`	N/A	`MERGE PARTITIONS`	N/A[1]
Modifying Default Attributes	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`
Modifying Real Attributes of Partitions	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`
Modifying List Partitions: Adding Values	N/A	N/A	N/A	`MODIFY PARTITION ... ADD VALUES`	N/A
Modifying List Partitions: Dropping Values	N/A	N/A	N/A	`MODIFY PARTITION ... DROP VALUES`	N/A
Moving Partitions	`MOVE PARTITION`	`MOVE PARTITION`	`MOVE PARTITION`	`MOVE PARTITION`	`MOVE PARTITION`

Table 3–1 (Cont.) ALTER TABLE Maintenance Operations for Table Partitions

Maintenance Operation	Range Composite Range-*	Interval Composite Interval-*	Hash	List Composite List-*	Reference
Renaming Partitions	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION`
Splitting Partitions	`SPLIT PARTITION`	`SPLIT PARTITION`	N/A	`SPLIT PARTITION`	N/A[1]
Truncating Partitions	`TRUNCATE PARTITION`	`TRUNCATE PARTITION`	`TRUNCATE PARTITION`	`TRUNCATE PARTITION`	`TRUNCATE PARTITION`

[1] These operations cannot be performed on reference-partitioned tables. If performed on a parent table, then these operations will cascade to all descendant tables.

Table 3–2 ALTER TABLE Maintenance Operations for Table Subpartitions

Maintenance Operation	Composite *-Range	Composite *-Hash	Composite *-List
Adding Partitions	`MODIFY PARTITION ... ADD SUBPARTITION`	`MODIFY PARTITION ... ADD SUBPARTITION`	`MODIFY PARTITION ... ADD SUBPARTITION`
Coalescing Partitions	N/A	`MODIFY PARTITION ... COALESCE SUBPARTITION`	N/A
Dropping Partitions	`DROP SUBPARTITION`	N/A	`DROP SUBPARTITION`
Exchanging Partitions	`EXCHANGE SUBPARTITION`	N/A	`EXCHANGE SUBPARTITION`
Merging Partitions	`MERGE SUBPARTITIONS`	N/A	`MERGE SUBPARTITIONS`
Modifying Default Attributes	`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`	`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`	`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`
Modifying Real Attributes of Partitions	`MODIFY SUBPARTITION`	`MODIFY SUBPARTITION`	`MODIFY SUBPARTITION`
Modifying List Partitions: Adding Values	N/A	N/A	`MODIFY SUBPARTITION ... ADD VALUES`
Modifying List Partitions: Dropping Values	N/A	N/A	`MODIFY SUBPARTITION ... DROP VALUES`
Modifying a Subpartition Template	`SET SUBPARTITION TEMPLATE`	`SET SUBPARTITION TEMPLATE`	`SET SUBPARTITION TEMPLATE`
Moving Partitions	`MOVE SUBPARTITION`	`MOVE SUBPARTITION`	`MOVE SUBPARTITION`
Renaming Partitions	`RENAME SUBPARTITION`	`RENAME SUBPARTITION`	`RENAME SUBPARTITION`
Splitting Partitions	`SPLIT SUBPARTITION`	N/A	`SPLIT SUBPARTITION`
Truncating Partitions	`TRUNCATE SUBPARTITION`	`TRUNCATE SUBPARTITION`	`TRUNCATE SUBPARTITION`

Note: The first time you use table compression to introduce a compressed partition into a partitioned table that has bitmap indexes and that currently contains only uncompressed partitions, you must do the following:

- Either drop all existing bitmap indexes and bitmap index partitions, or mark them `UNUSABLE`.
- Set the table compression attribute.
- Rebuild the indexes.

These actions are independent of whether any partitions contain data and of the operation that introduces the compressed partition.

This does not apply to partitioned tables with B-tree indexes or to partitioned index-organized tables.

Table 3–3 lists maintenance operations that can be performed on index partitions, and indicates on which type of index (global or local) they can be performed. The `ALTER INDEX` clause used for the maintenance operation is shown.

Global indexes do not reflect the structure of the underlying table. If partitioned, they can be partitioned by range or hash. Partitioned global indexes share some, but not all, of the partition maintenance operations that can be performed on partitioned tables.

Because local indexes reflect the underlying structure of the table, partitioning is maintained automatically when table partitions and subpartitions are affected by maintenance activity. Therefore, partition maintenance on local indexes is less necessary and there are fewer options.

Table 3–3 ALTER INDEX Maintenance Operations for Index Partitions

Maintenance Operation	**Type of Index**	**Type of Index Partitioning**		
		Range	**Hash and List**	**Composite**
Adding Index Partitions	Global	–	`ADD PARTITION` (hash only)	-
	Local	N/A	N/A	N/A
Dropping Index Partitions	Global	`DROP PARTITION`	-	-
	Local	N/A	N/A	N/A
Modifying Default Attributes of Index Partitions	Global	`MODIFY DEFAULT ATTRIBUTES`	-	-

	Local	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`
				`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`

Modifying Real Attributes of Index Partitions	Global	`MODIFY PARTITION`	-	-

	Local	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`
				`MODIFY SUBPARTITION`

Rebuilding Index Partitions	Global	`REBUILD PARTITION`	-	-
	Local	`REBUILD PARTITION`	`REBUILD PARTITION`	`REBUILD SUBPARTITION`

Table 3–3 (Cont.) ALTER INDEX Maintenance Operations for Index Partitions

Maintenance Operation	**Type of Index**	**Type of Index Partitioning**		
		Range	**Hash and List**	**Composite**
Renaming Index Partitions	Global	`RENAME PARTITION`	-	-
	Local	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION` `RENAME SUBPARTITION`
Splitting Index Partitions	Global	`SPLIT PARTITION`	-	-
	Local	N/A	N/A	N/A

Note: The following sections discuss maintenance operations on partitioned tables. Where the usability of indexes or index partitions affected by the maintenance operation is discussed, consider the following:

- Only indexes and index partitions that are *not* empty are candidates for being marked `UNUSABLE`. If they are empty, the `USABLE/UNUSABLE` status is left unchanged.
- Only indexes or index partitions with `USABLE` status are updated by subsequent DML.

Updating Indexes Automatically

Before discussing the individual maintenance operations for partitioned tables and indexes, it is important to discuss the effects of the `UPDATE INDEXES` clause that can be specified in the `ALTER TABLE` statement.

By default, many table maintenance operations on partitioned tables invalidate (mark `UNUSABLE`) the corresponding indexes or index partitions. You must then rebuild the entire index or, in the case of a global index, each of its partitions. The database lets you override this default behavior if you specify `UPDATE INDEXES` in your `ALTER TABLE` statement for the maintenance operation. Specifying this clause tells the database to update the index at the time it executes the maintenance operation DDL statement. This provides the following benefits:

- The index is updated in conjunction with the base table operation. You are not required to later and independently rebuild the index.
- The index is more highly available, because it does not get marked `UNUSABLE`. The index remains available even while the partition DDL is executing and it can be used to access unaffected partitions in the table.
- You need not look up the names of all invalid indexes to rebuild them.

Optional clauses for local indexes let you specify physical and storage characteristics for updated local indexes and their partitions.

- You can specify physical attributes, tablespace storage, and logging for each partition of each local index. Alternatively, you can specify only the `PARTITION` keyword and let the database update the partition attributes as follows:
  - For operations on a single table partition (such as `MOVE PARTITION` and `SPLIT PARTITION`), the corresponding index partition inherits the attributes of the affected index partition. The database does not generate names for new

index partitions, so any new index partitions resulting from this operation inherit their names from the corresponding new table partition.

  – For MERGE PARTITION operations, the resulting local index partition inherits its name from the resulting table partition and inherits its attributes from the local index.

- For a composite-partitioned index, you can specify tablespace storage for each subpartition.

See Also: the *update_all_indexes_clause* of ALTER TABLE for the syntax for updating indexes

The following operations support the UPDATE INDEXES clause:

- ADD PARTITION | SUBPARTITION
- COALESCE PARTITION | SUBPARTITION
- DROP PARTITION | SUBPARTITION
- EXCHANGE PARTITION | SUBPARTITION
- MERGE PARTITION | SUBPARTITION
- MOVE PARTITION | SUBPARTITION
- SPLIT PARTITION | SUBPARTITION
- TRUNCATE PARTITION | SUBPARTITION

SKIP_UNUSABLE_INDEXES Initialization Parameter

SKIP_UNUSABLE_INDEXES is an initialization parameter with a default value of TRUE. This setting disables error reporting of indexes and index partitions marked UNUSABLE. If you do not want the database to choose an alternative execution plan to avoid the unusable elements, then you should set this parameter to FALSE.

Considerations when Updating Indexes Automatically

The following implications are worth noting when you specify UPDATE INDEXES:

- The partition DDL statement takes longer to execute, because indexes that were previously marked UNUSABLE are updated. However, you must compare this increase with the time it takes to execute DDL without updating indexes, and then rebuild all indexes. A rule of thumb is that it is faster to update indexes if the size of the partition is less that 5% of the size of the table.
- The DROP, TRUNCATE, and EXCHANGE operations are no longer fast operations. Again, you must compare the time it takes to do the DDL and then rebuild all indexes.
- When you update a table with a global index:
  – The index is updated in place. The updates to the index are logged, and redo and undo records are generated. In contrast, if you rebuild an entire global index, you can do so in NOLOGGING mode.
  – Rebuilding the entire index manually creates a more efficient index, because it is more compact with space better utilized.
- The UPDATE INDEXES clause is not supported for index-organized tables. However, the UPDATE GLOBAL INDEXES clause may be used with DROP PARTITION, TRUNCATE PARTITION, and EXCHANGE PARTITION operations to keep the global indexes on index-organized tables usable. For the remaining

operations in the above list, global indexes on index-organized tables remain usable. In addition, local index partitions on index-organized tables remain usable after a MOVE PARTITION operation.

Adding Partitions

This section describes how to manually add new partitions to a partitioned table and explains why partitions cannot be specifically added to most partitioned indexes.

Adding a Partition to a Range-Partitioned Table

Use the ALTER TABLE ... ADD PARTITION statement to add a new partition to the "high" end (the point after the last existing partition). To add a partition at the beginning or in the middle of a table, use the SPLIT PARTITION clause.

For example, consider the table, sales, which contains data for the current month in addition to the previous 12 months. On January 1, 1999, you add a partition for January, which is stored in tablespace tsx.

```
ALTER TABLE sales
      ADD PARTITION jan99 VALUES LESS THAN ( '01-FEB-1999' )
      TABLESPACE tsx;
```

Local and global indexes associated with the range-partitioned table remain usable.

Adding a Partition to a Hash-Partitioned Table

When you add a partition to a hash-partitioned table, the database populates the new partition with rows rehashed from an existing partition (selected by the database) as determined by the hash function. As a result, if the table contains data, then it may take some time to add a hash partition.

The following statements show two ways of adding a hash partition to table scubagear. Choosing the first statement adds a new hash partition whose partition name is system generated, and which is placed in the default tablespace. The second statement also adds a new hash partition, but that partition is explicitly named p_named and is created in tablespace gear5.

```
ALTER TABLE scubagear ADD PARTITION;

ALTER TABLE scubagear
      ADD PARTITION p_named TABLESPACE gear5;
```

Indexes may be marked UNUSABLE as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify UPDATE INDEXES as part of the ALTER TABLE statement: ▪ The local indexes for the new partition, and for the existing partition from which rows were redistributed, are marked UNUSABLE and must be rebuilt. ▪ All global indexes, or all partitions of partitioned global indexes, are marked UNUSABLE and must be rebuilt.
Index-organized	▪ For local indexes, the behavior is the same as for heap tables. ▪ All global indexes remain usable.

Adding a Partition to a List-Partitioned Table

The following statement illustrates how to add a new partition to a list-partitioned table. In this example physical attributes and NOLOGGING are specified for the partition being added.

```
ALTER TABLE q1_sales_by_region
   ADD PARTITION q1_nonmainland VALUES ('HI', 'PR')
      STORAGE (INITIAL 20K NEXT 20K) TABLESPACE tbs_3
      NOLOGGING;
```

Any value in the set of literal values that describe the partition being added must not exist in any of the other partitions of the table.

You cannot add a partition to a list-partitioned table that has a default partition, but you can split the default partition. By doing so, you effectively create a new partition defined by the values that you specify, and a second partition that remains the default partition.

Local and global indexes associated with the list-partitioned table remain usable.

Adding a Partition to an Interval-Partitioned Table

You cannot explicitly add a partition to an interval-partitioned table unless you first lock the partition, which triggers the creation of the partition. The database automatically creates a partition for an interval when data for that interval is inserted. In general, you only need to explicitly create interval partitions for a partition exchange load scenario.

To change the interval for future partitions, use the SET INTERVAL clause of the ALTER TABLE statement. This clause changes the interval for partitions beyond the current highest boundary of all materialized interval partitions.

You also use the SET INTERVAL clause to migrate an existing range partitioned or range-* composite partitioned table into an interval or interval-* partitioned table. If you want to disable the creation of future interval partitions, and effectively revert back to a range-partitioned table, then use an empty value in the SET INTERVAL clause. Created interval partitions will then be transformed into range partitions with their current high values.

To increase the interval for date ranges, then you need to ensure that you are at a relevant boundary for the new interval. For example, if the highest interval partition boundary in your daily interval partitioned table transactions is January 30, 2007 and you want to change to a monthly partition interval, then the following statement results in an error:

```
ALTER TABLE transactions SET INTERVAL (NUMTOYMINTERVAL(1,'MONTH');

ORA-14767: Cannot specify this interval with existing high bounds
```

You need to create another daily partition with a high bound of February 1, 2007 in order to successfully change to a monthly interval:

```
LOCK TABLE transactions PARTITION FOR(TO_DATE('31-JAN-2007','dd-MON-yyyy') IN
SHARE MODE;

ALTER TABLE transactions SET INTERVAL (NUMTOYMINTERVAL(1,'MONTH');
```

The lower partitions of an interval-partitioned table are range partitions. You can split range partitions in order to add more partitions in the range portion of the interval-partitioned table.

In order to disable interval partitioning on the `transactions` table, use:

```
ALTER TABLE transactions SET INTERVAL ();
```

Adding Partitions to a Composite [Range | List | Interval]-Hash Partitioned Table

Partitions can be added at both the partition level and at the hash subpartition level.

Adding a Partition to a [Range | List | Interval]-Hash Partitioned Table Adding a new partition to a [range | list | interval]-hash partitioned table is as described previously. For an interval-hash partitioned table, interval partitions are automatically created. You can specify a `SUBPARTITIONS` clause that lets you add a specified number of subpartitions, or a `SUBPARTITION` clause for naming specific subpartitions. If no `SUBPARTITIONS` or `SUBPARTITION` clause is specified, then the partition inherits table level defaults for subpartitions. For an interval-hash partitioned table, you can only add subpartitions to range or interval partitions that have been materialized.

This example adds a range partition `q1_2000` to the range-hash partitioned table `sales`, which will be populated with data for the first quarter of the year 2000. There are eight subpartitions stored in tablespace `tbs5`. The subpartitions cannot be set explicitly to use table compression. Subpartitions inherit the compression attribute from the partition level and are stored in a compressed form in this example:

```
ALTER TABLE sales ADD PARTITION q1_2000
      VALUES LESS THAN (2000, 04, 01) COMPRESS
      SUBPARTITIONS 8 STORE IN tbs5;
```

Adding a Subpartition to a [Range | List | Interval]-Hash Partitioned Table You use the `MODIFY PARTITION` ... `ADD SUBPARTITION` clause of the `ALTER TABLE` statement to add a hash subpartition to a [range | list | interval]-hash partitioned table. The newly added subpartition is populated with rows rehashed from other subpartitions of the same partition as determined by the hash function. For an interval-hash partitioned table, you can only add subpartitions to range or interval partitions that have been materialized.

In the following example, a new hash subpartition `us_loc5`, stored in tablespace `us1`, is added to range partition `locations_us` in table `diving`.

```
ALTER TABLE diving MODIFY PARTITION locations_us
      ADD SUBPARTITION us_locs5 TABLESPACE us1;
```

Index subpartitions corresponding to the added and rehashed subpartitions must be rebuilt unless you specify `UPDATE INDEXES`.

Adding Partitions to a Composite [Range | List | Interval]-List Partitioned Table

Partitions can be added at both the partition level and at the list subpartition level.

Adding a Partition to a [Range | List | Interval]-List Partitioned Table Adding a new partition to a [range | list | interval]-list partitioned table is as described previously. The database automatically creates interval partitions as data for a specific interval is inserted. You can specify `SUBPARTITION` clauses for naming and providing value lists for the subpartitions. If no `SUBPARTITION` clauses are specified, then the partition inherits the subpartition template. If there is no subpartition template, then a single default subpartition is created.

The following statement adds a new partition to the `quarterly_regional_sales` table that is partitioned by the range-list method. Some new physical attributes are specified for this new partition while table-level defaults are inherited for those that are not specified.

```
ALTER TABLE quarterly_regional_sales
   ADD PARTITION q1_2000 VALUES LESS THAN (TO_DATE('1-APR-2000','DD-MON-YYYY'))
      STORAGE (INITIAL 20K NEXT 20K) TABLESPACE ts3 NOLOGGING
         (
          SUBPARTITION q1_2000_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q1_2000_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q1_2000_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q1_2000_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q1_2000_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q1_2000_southcentral VALUES ('OK', 'TX')
         );
```

Adding a Subpartition to a [Range | List | Interval]-List Partitioned Table You use the `MODIFY PARTITION ... ADD SUBPARTITION` clause of the `ALTER TABLE` statement to add a list subpartition to a [range | list | interval]-list partitioned table. For an interval-list partitioned table, you can only add subpartitions to range or interval partitions that have been materialized.

The following statement adds a new subpartition to the existing set of subpartitions in the range-list partitioned table `quarterly_regional_sales`. The new subpartition is created in tablespace `ts2`.

```
ALTER TABLE quarterly_regional_sales
   MODIFY PARTITION q1_1999
      ADD SUBPARTITION q1_1999_south
         VALUES ('AR','MS','AL') tablespace ts2;
```

Adding Partitions to a Composite [Range | List | Interval]-Range Partitioned Table

Partitions can be added at both the partition level and at the range subpartition level.

Adding a Partition to a [Range | List | Interval]-Range Partitioned Table Adding a new partition to a [range | list | interval]-range partitioned table is as described previously. The database automatically creates interval partitions for an interval-range partitioned table when data is inserted in a specific interval. You can specify a `SUBPARTITION` clause for naming and providing ranges for specific subpartitions. If no `SUBPARTITION` clause is specified, then the partition inherits the subpartition template specified at the table level. If there is no subpartition template, then a single subpartition with a maximum value of `MAXVALUE` is created.

This example adds a range partition `p_2007_jan` to the range-range partitioned table `shipments`, which will be populated with data for the shipments ordered in January 2007. There are three subpartitions. Subpartitions inherit the compression attribute from the partition level and are stored in a compressed form in this example:

```
ALTER TABLE shipments
   ADD PARTITION p_2007_jan
      VALUES LESS THAN (TO_DATE('01-FEB-2007','dd-MON-yyyy')) COMPRESS
      ( SUBPARTITION p07_jan_e VALUES LESS THAN (TO_
DATE('15-FEB-2007','dd-MON-yyyy'))
      , SUBPARTITION p07_jan_a VALUES LESS THAN (TO_
DATE('01-MAR-2007','dd-MON-yyyy'))
      , SUBPARTITION p07_jan_l VALUES LESS THAN (TO_
DATE('01-APR-2007','dd-MON-yyyy'))
      ) ;
```

Adding a Subpartition to a [Range | List | Interval]-Range Partitioned Table You use the `MODIFY PARTITION ... ADD SUBPARTITION` clause of the `ALTER TABLE` statement to add a range subpartition to a [range | list | interval]-range partitioned table. For an interval-range partitioned table, you can only add partitions to range or interval partitions that have already been materialized.

The following example adds a range subpartition to the `shipments` table that will contain all values with an `order_date` in January 2007 and a `delivery_date` on or after April 1, 2007.

```
ALTER TABLE shipments
   MODIFY PARTITION p_2007_jan
      ADD SUBPARTITION p07_jan_vl VALUES LESS THAN (MAXVALUE) ;
```

Adding a Partition or Subpartition to a Reference-Partitioned Table

A partition or subpartition can be added to a parent table in a reference partition definition just as partitions and subpartitions can be added to a range, hash, list, or composite partitioned table. The add operation will automatically cascade to any descendant reference partitioned tables. The `DEPENDENT TABLES` clause can be used to set specific properties for dependent tables when you add partitions or subpartitions to a master table.

See Also: *Oracle Database SQL Language Reference*

Adding Index Partitions

You cannot explicitly add a partition to a local index. Instead, a new partition is added to a local index only when you add a partition to the underlying table. Specifically, when there is a local index defined on a table and you issue the `ALTER TABLE` statement to add a partition, a matching partition is also added to the local index. The database assigns names and default physical storage attributes to the new index partitions, but you can rename or alter them after the `ADD PARTITION` operation is complete.

You can effectively specify a new tablespace for an index partition in an `ADD PARTITION` operation by first modifying the default attributes for the index. For example, assume that a local index, `q1_sales_by_region_locix`, was created for list partitioned table `q1_sales_by_region`. If before adding the new partition `q1_nonmainland`, as shown in "Adding a Partition to a List-Partitioned Table" on page 3-33, you had issued the following statement, then the corresponding index partition would be created in tablespace `tbs_4`.

```
ALTER INDEX q1_sales_by_region_locix
   MODIFY DEFAULT ATTRIBUTES TABLESPACE tbs_4;
```

Otherwise, it would be necessary for you to use the following statement to move the index partition to `tbs_4` after adding it:

```
ALTER INDEX q1_sales_by_region_locix
   REBUILD PARTITION q1_nonmainland TABLESPACE tbs_4;
```

You can add a partition to a hash-partitioned global index using the `ADD PARTITION` syntax of `ALTER INDEX`. The database adds hash partitions and populates them with index entries rehashed from an existing hash partition of the index, as determined by the hash function. The following statement adds a partition to the index `hgidx` shown in "Creating a Hash-Partitioned Global Index" on page 3-5:

```
ALTER INDEX hgidx ADD PARTITION p5;
```

You cannot add a partition to a range-partitioned global index, because the highest partition always has a partition bound of `MAXVALUE`. If you want to add a new highest partition, use the `ALTER INDEX` ... `SPLIT PARTITION` statement.

Coalescing Partitions

Coalescing partitions is a way of reducing the number of partitions in a hash-partitioned table or index, or the number of subpartitions in a *-hash partitioned table. When a hash partition is coalesced, its contents are redistributed into one or more remaining partitions determined by the hash function. The specific partition that is coalesced is selected by the database, and is dropped after its contents have been redistributed. If you coalesce a hash partition or subpartition in the parent table of a reference-partitioned table definition, then the reference-partitioned table automatically inherits the new partitioning definition.

Index partitions may be marked `UNUSABLE` as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: ▪ Any local index partition corresponding to the selected partition is also dropped. Local index partitions corresponding to the one or more absorbing partitions are marked `UNUSABLE` and must be rebuilt. ▪ All global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	▪ Some local indexes are marked `UNUSABLE` as noted for heap indexes. ▪ All global indexes remain usable.

Coalescing a Partition in a Hash-Partitioned Table

The `ALTER TABLE` ... `COALESCE PARTITION` statement is used to coalesce a partition in a hash-partitioned table. The following statement reduces by one the number of partitions in a table by coalescing a partition.

```
ALTER TABLE ouu1
     COALESCE PARTITION;
```

Coalescing a Subpartition in a *-Hash Partitioned Table

The following statement distributes the contents of a subpartition of partition `us_locations` into one or more remaining subpartitions (determined by the hash function) of the same partition. Note that for an interval-partitioned table, you can only coalesce hash subpartitions of materialized range or interval partitions. Basically, this operation is the inverse of the `MODIFY PARTITION` ... `ADD SUBPARTITION` clause discussed in "Adding a Subpartition to a [Range | List | Interval]-Hash Partitioned Table" on page 3-34.

```
ALTER TABLE diving MODIFY PARTITION us_locations
     COALESCE SUBPARTITION;
```

Coalescing Hash-partitioned Global Indexes

You can instruct the database to reduce by one the number of index partitions in a hash-partitioned global index using the `COALESCE PARTITION` clause of `ALTER`

`INDEX`. The database selects the partition to coalesce based on the requirements of the hash partition. The following statement reduces by one the number of partitions in the `hgidx` index, created in "Creating a Hash-Partitioned Global Index" on page 3-5:

```
ALTER INDEX hgidx COALESCE PARTITION;
```

Dropping Partitions

You can drop partitions from range, interval, list, or composite *-[range | list] partitioned tables. For interval partitioned tables, you can only drop range or interval partitions that have been materialized. For hash-partitioned tables, or hash subpartitions of composite *-hash partitioned tables, you must perform a coalesce operation instead.

You cannot drop a partition from a reference-partitioned table. Instead, a drop operation on a parent table will cascade to all descendant tables.

Dropping Table Partitions

Use one of the following statements to drop a table partition or subpartition:

- `ALTER TABLE` ... `DROP PARTITION` to drop a table partition
- `ALTER TABLE` ... `DROP SUBPARTITION` to drop a subpartition of a composite *-[range | list] partitioned table

If you want to preserve the data in the partition, then use the `MERGE PARTITION` statement instead of the `DROP PARTITION` statement.

If local indexes are defined for the table, then this statement also drops the matching partition or subpartitions from the local index. All global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` unless either of the following is true:

- You specify `UPDATE INDEXES` (Cannot be specified for index-organized tables. Use `UPDATE GLOBAL INDEXES` instead.)
- The partition being dropped or its subpartitions are empty

Note:

- You cannot drop the only partition in a table. Instead, you must drop the table.
- You cannot drop the highest range partition in the range-partitioned section of an interval-partitioned or interval-* composite partitioned table.

The following sections contain some scenarios for dropping table partitions.

Dropping a Partition from a Table that Contains Data and Global Indexes If the partition contains data and one or more global indexes are defined on the table, then use one of the following methods to drop the table partition.

Method 1

Leave the global indexes in place during the `ALTER TABLE` ... `DROP PARTITION` statement. Afterward, you must rebuild any global indexes (whether partitioned or not) because the index (or index partitions) will have been marked `UNUSABLE`. The following statements provide an example of dropping partition dec98 from the sales table, then rebuilding its global non-partitioned index.

```
ALTER TABLE sales DROP PARTITION dec98;
ALTER INDEX sales_area_ix REBUILD;
```

If index `sales_area_ix` were a range-partitioned global index, then all partitions of the index would require rebuilding. Further, it is not possible to rebuild all partitions of an index in one statement. You must issue a separate `REBUILD` statement for each partition in the index. The following statements rebuild the index partitions `jan99_ix`, `feb99_ix`, `mar99_ix`, ..., `dec99_ix`.

```
ALTER INDEX sales_area_ix REBUILD PARTITION jan99_ix;
ALTER INDEX sales_area_ix REBUILD PARTITION feb99_ix;
ALTER INDEX sales_area_ix REBUILD PARTITION mar99_ix;
...
ALTER INDEX sales_area_ix REBUILD PARTITION dec99_ix;
```

This method is most appropriate for large tables where the partition being dropped contains a significant percentage of the total data in the table.

Method 2

Issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE` ... `DROP PARTITION` statement. The `DELETE` statement updates the global indexes.

For example, to drop the first partition, issue the following statements:

```
DELETE FROM sales partition (dec98);
ALTER TABLE sales DROP PARTITION dec98;
```

This method is most appropriate for small tables, or for large tables when the partition being dropped contains a small percentage of the total data in the table.

Method 3

Specify `UPDATE INDEXES` in the `ALTER TABLE` statement. Doing so causes the global index to be updated at the time the partition is dropped.

```
ALTER TABLE sales DROP PARTITION dec98
     UPDATE INDEXES;
```

Dropping a Partition Containing Data and Referential Integrity Constraints If a partition contains data and the table has referential integrity constraints, choose either of the following methods to drop the table partition. This table has a local index only, so it is not necessary to rebuild any indexes.

Method 1

If there is no data referencing the data in the partition you want to drop, then you can disable the integrity constraints on the referencing tables, issue the `ALTER TABLE` ... `DROP PARTITION` statement, then re-enable the integrity constraints.

This method is most appropriate for large tables where the partition being dropped contains a significant percentage of the total data in the table. If there is still data referencing the data in the partition to be dropped, then make sure to remove all the referencing data in order to be able to re-enable the referential integrity constraints.

Method 2

If there is data in the referencing tables, then you can issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE` ... `DROP PARTITION` statement. The `DELETE` statement enforces referential integrity constraints, and also fires triggers and generates redo and undo logs. The delete can succeed if you created the constraints with the `ON DELETE CASCADE` option, deleting all rows from referencing tables as well.

```
DELETE FROM sales partition (dec94);
ALTER TABLE sales DROP PARTITION dec94;
```

This method is most appropriate for small tables or for large tables when the partition being dropped contains a small percentage of the total data in the table.

Dropping Interval Partitions

You can drop interval partitions in an interval-partitioned table. This operation will drop the data for the interval only and leave the interval definition in tact. If data is inserted in the interval just dropped, then the database will again create an interval partition.

You can also drop range partitions in an interval-partitioned table. The rules for dropping a range partition in an interval-partitioned table follow the rules for dropping a range partition in a range-partitioned table. If you drop a range partition in the middle of a set of range partitions, then the lower boundary for the next range partition shifts to the lower boundary of the range partition you just dropped. You cannot drop the highest range partition in the range-partitioned section of an interval-partitioned table.

The following example drops the September 2007 interval partition from the `sales` table. There are only local indexes so no indexes will be invalidated.

```
ALTER TABLE sales DROP PARTITION FOR(TO_DATE('01-SEP-2007','dd-MON-yyyy'));
```

Dropping Index Partitions

You cannot explicitly drop a partition of a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.

If a global index partition is empty, then you can explicitly drop it by issuing the `ALTER INDEX` ... `DROP PARTITION` statement. But, if a global index partition contains data, then dropping the partition causes the next highest partition to be marked `UNUSABLE`. For example, you would like to drop the index partition P1, and P2 is the next highest partition. You must issue the following statements:

```
ALTER INDEX npr DROP PARTITION P1;
ALTER INDEX npr REBUILD PARTITION P2;
```

Note: You cannot drop the highest partition in a global index.

Exchanging Partitions

You can convert a partition (or subpartition) into a non-partitioned table, and a non-partitioned table into a partition (or subpartition) of a partitioned table by exchanging their data segments. You can also convert a hash-partitioned table into a partition of a composite *-hash partitioned table, or convert the partition of a composite *-hash partitioned table into a hash-partitioned table. Similarly, you can convert a [range | list]-partitioned table into a partition of a composite *-[range | list] partitioned table, or convert a partition of the composite *-[range | list] partitioned table into a [range | list]-partitioned table.

Exchanging table partitions is most useful when you have an application using non-partitioned tables that you want to convert to partitions of a partitioned table. For example, in data warehousing environments, exchanging partitions facilitates high-speed data loading of new, incremental data into an already existing partitioned table. Generically, OLTP as well as data warehousing environments benefit from exchanging old data partitions out of a partitioned table. The data is purged from the partitioned table without actually being deleted and can be archived separately afterwards.

When you exchange partitions, logging attributes are preserved. You can optionally specify if local indexes are also to be exchanged (`INCLUDING INDEXES` clause), and if rows are to be validated for proper mapping (`WITH VALIDATION` clause).

Note: When you specify `WITHOUT VALIDATION` for the exchange partition operation, this is normally a fast operation because it involves only data dictionary updates. However, if the table or partitioned table involved in the exchange operation has a primary key or unique constraint enabled, then the exchange operation will be performed as if `WITH VALIDATION` were specified in order to maintain the integrity of the constraints.

To avoid the overhead of this validation activity, issue the following statement for each constraint before doing the exchange partition operation:

```
ALTER TABLE table_name
     DISABLE CONSTRAINT constraint_name KEEP INDEX
```

Then, enable the constraints after the exchange.

If you specify `WITHOUT VALIDATION`, then you have to make sure that the data to be exchanged belongs in the partition you exchange.

Unless you specify `UPDATE INDEXES`, the database marks `UNUSABLE` the global indexes or all global index partitions on the table whose partition is being exchanged. Global indexes or global index partitions on the table being exchanged remain invalidated. (You cannot use `UPDATE INDEXES` for index-organized tables. Use `UPDATE GLOBAL INDEXES` instead.)

See Also: "Viewing Information About Partitioned Tables and Indexes" on page 3-69

Exchanging a Range, Hash, or List Partition

To exchange a partition of a range, hash, or list-partitioned table with a non-partitioned table, or the reverse, use the `ALTER TABLE ... EXCHANGE PARTITION` statement. An example of converting a partition into a non-partitioned table follows. In this example, table `stocks` can be range, hash, or list partitioned.

```
ALTER TABLE stocks
    EXCHANGE PARTITION p3 WITH TABLE stock_table_3;
```

Exchanging a Partition of an Interval Partitioned Table

You can exchange interval partitions in an interval-partitioned table. However, you have to make sure the interval partition has been created before you can exchange the partition. You can let the database create the partition by locking the interval partition.

The following example shows a partition exchange for the `interval_sales` table, interval-partitioned using monthly partitions as of January 1, 2004. This example shows how to add data for June 2007 to the table using partition exchange load. Assume there are only local indexes on the `interval_sales` table, and equivalent indexes have been created on the `interval_sales_june_2007` table.

```
LOCK TABLE interval_sales
PARTITION FOR (TO_DATE('01-JUN-2007','dd-MON-yyyy'))
IN SHARE MODE;

ALTER TABLE interval_sales
EXCHANGE PARTITION FOR (TO_DATE('01-JUN-2007','dd-MON-yyyy'))
WITH TABLE interval_sales_jun_2007
INCLUDING INDEXES;
```

Note the use of the `FOR` syntax to identify a partition that was system-generated. The partition name can be used by querying the `*_TAB_PARTITIONS` data dictionary view to find out the system-generated partition name.

Exchanging a Partition of a Reference Partitioned Table

You can exchange partitions in a reference-partitioned table, but you have to make sure that data you reference is available in the respective partition in the parent table.

The following example shows a partition exchange load scenario for the range-partitioned `orders` table, and the reference partitioned `order_items` table. Note that the data in the `order_items_dec_2006` table only contains order item data for orders with an `order_date` in December 2006.

```
ALTER TABLE orders
EXCHANGE PARTITION p_2006_dec
WITH TABLE orders_dec_2006
UPDATE GLOBAL INDEXES;

ALTER TABLE order_items_dec_2006
ADD CONSTRAINT order_items_dec_2006_fk
FOREIGN KEY (order_id)
REFERENCES orders(order_id) ;

ALTER TABLE order_items
EXCHANGE PARTITION p_2006_dec
WITH TABLE order_items_dec_2006;
```

Note that you have to use the `UPDATE GLOBAL INDEXES` or `UPDATE INDEXES` on the exchange partition of the parent table in order for the primary key index to remain usable. Note also that you have to create or enable the foreign key constraint on the `order_items_dec_2006` table in order for the partition exchange on the reference-partitioned table to succeed.

Exchanging a Partition of a Table with Virtual Columns

You can exchange partitions in the presence of virtual columns. In order for a partition exchange on a partitioned table with virtual columns to succeed, you have to create a table that matches the definition of all non-virtual columns in a single partition of the partitioned table. You do not need to include the virtual column definitions, unless constraints or indexes have been defined on the virtual column.

In this case, you have to include the virtual column definition in order to match the partitioned table's constraint and index definitions. This scenario also applies to virtual column-based partitioned tables.

Exchanging a Hash-Partitioned Table with a *-Hash Partition

In this example, you are exchanging a whole hash-partitioned table, with all of its partitions, with the partition of a *-hash partitioned table and all of its hash subpartitions. The following example illustrates this concept for a range-hash partitioned table.

First, create a hash-partitioned table:

```
CREATE TABLE t1 (i NUMBER, j NUMBER)
     PARTITION BY HASH(i)
       (PARTITION p1, PARTITION p2);
```

Populate the table, then create a range-hash partitioned table as shown:

```
CREATE TABLE t2 (i NUMBER, j NUMBER)
     PARTITION BY RANGE(j)
     SUBPARTITION BY HASH(i)
        (PARTITION p1 VALUES LESS THAN (10)
            SUBPARTITION t2_pls1
            SUBPARTITION t2_pls2,
         PARTITION p2 VALUES LESS THAN (20)
            SUBPARTITION t2_p2s1
            SUBPARTITION t2_p2s2));
```

It is important that the partitioning key in table `t1` is the same as the subpartitioning key in table `t2`.

To migrate the data in `t1` to `t2`, and validate the rows, use the following statement:

```
ALTER TABLE t2 EXCHANGE PARTITION p1 WITH TABLE t1
     WITH VALIDATION;
```

Exchanging a Subpartition of a *-Hash Partitioned Table

Use the `ALTER TABLE` ... `EXCHANGE SUBPARTITION` statement to convert a hash subpartition of a *-hash partitioned table into a non-partitioned table, or the reverse. The following example converts the subpartition `q3_1999_s1` of table `sales` into the non-partitioned table `q3_1999`. Local index partitions are exchanged with corresponding indexes on `q3_1999`.

```
ALTER TABLE sales EXCHANGE SUBPARTITION q3_1999_s1
      WITH TABLE q3_1999 INCLUDING INDEXES;
```

Exchanging a List-Partitioned Table with a *-List Partition

The semantics of the `ALTER TABLE` ... `EXCHANGE PARTITION` statement are the same as described previously in "Exchanging a Hash-Partitioned Table with a *-Hash Partition" on page 3-43. The example below shows an exchange partition scenario for a list-list partitioned table.

```
CREATE TABLE customers_apac
( id            NUMBER
, name          VARCHAR2(50)
, email         VARCHAR2(100)
, region        VARCHAR2(4)
, credit_rating VARCHAR2(1)
)
PARTITION BY LIST (credit_rating)
( PARTITION poor VALUES ('P')
, PARTITION mediocre VALUES ('C')
, PARTITION good VALUES ('G')
, PARTITION excellent VALUES ('E')
);
```

Populate the table with APAC customers. Then create a list-list partitioned table:

```
CREATE TABLE customers
( id            NUMBER
, name          VARCHAR2(50)
, email         VARCHAR2(100)
, region        VARCHAR2(4)
, credit_rating VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY LIST (credit_rating)
SUBPARTITION TEMPLATE
( SUBPARTITION poor VALUES ('P')
, SUBPARTITION mediocre VALUES ('C')
, SUBPARTITION good VALUES ('G')
, SUBPARTITION excellent VALUES ('E')
)
(PARTITION americas VALUES ('AMER')
, PARTITION emea VALUES ('EMEA')
, PARTITION apac VALUES ('APAC')
);
```

It is important that the partitioning key in the `customers_apac` table matches the subpartitioning key in the `customers` table.

Next, exchange the `apac` partition.

```
ALTER TABLE customers
EXCHANGE PARTITION apac
WITH TABLE customers_apac
WITH VALIDATION;
```

Exchanging a Subpartition of a *-List Partitioned Table

The semantics of the `ALTER TABLE` ... `EXCHANGE SUBPARTITION` are the same as described previously in "Exchanging a Subpartition of a *-Hash Partitioned Table" on page 3-43.

Exchanging a Range-Partitioned Table with a *-Range Partition

The semantics of the `ALTER TABLE` ... `EXCHANGE PARTITION` statement are the same as described previously in "Exchanging a Hash-Partitioned Table with a *-Hash Partition" on page 3-43. The example below shows the `orders` table, which is interval partitioned by `order_date`, and subpartitioned by range on `order_total`. The example shows how to exchange a single monthly interval with a range-partitioned table.

```
CREATE TABLE orders_mar_2007
( id          NUMBER
, cust_id     NUMBER
, order_date  DATE
, order_total NUMBER
)
PARTITION BY RANGE (order_total)
( PARTITION p_small VALUES LESS THAN (1000)
, PARTITION p_medium VALUES LESS THAN (10000)
, PARTITION p_large VALUES LESS THAN (100000)
, PARTITION p_extraordinary VALUES LESS THAN (MAXVALUE)
);
```

Populate the table with orders for March 2007. Then create an interval-range partitioned table:

```
CREATE TABLE orders
( id          NUMBER
, cust_id     NUMBER
, order_date  DATE
, order_total NUMBER
)
PARTITION BY RANGE (order_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
  SUBPARTITION BY RANGE (order_total)
  SUBPARTITION TEMPLATE
  ( SUBPARTITION p_small VALUES LESS THAN (1000)
  , SUBPARTITION p_medium VALUES LESS THAN (10000)
  , SUBPARTITION p_large VALUES LESS THAN (100000)
  , SUBPARTITION p_extraordinary VALUES LESS THAN (MAXVALUE)
  )
(PARTITION p_before_2007 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-
MON-yyyy')));
```

It is important that the partitioning key in the `orders_mar_2007` table matches the subpartitioning key in the `orders` table.

Next, exchange the partition. Note that since an interval partition is to be exchanged, the partition is first locked to ensure that the partition is created.

```
LOCK TABLE orders PARTITION FOR (TO_DATE('01-MAR-2007','dd-MON-yyyy'))
IN SHARE MODE;

ALTER TABLE orders
EXCHANGE PARTITION
FOR (TO_DATE('01-MAR-2007','dd-MON-yyyy'))
WITH TABLE orders_mar_2007
WITH VALIDATION;
```

Exchanging a Subpartition of a *-Range Partitioned Table

The semantics of the `ALTER TABLE` ... `EXCHANGE SUBPARTITION` are the same as described previously in "Exchanging a Subpartition of a *-Hash Partitioned Table" on page 3-43.

Merging Partitions

Use the `ALTER TABLE` ... `MERGE PARTITION` statement to merge the contents of two partitions into one partition. The two original partitions are dropped, as are any corresponding local indexes.

You cannot use this statement for a hash-partitioned table or for hash subpartitions of a composite *-hash partitioned table.

You cannot merge partitions for a reference-partitioned table. Instead, a merge operation on a parent table will cascade to all descendant tables. However, you can use the `DEPENDENT TABLES` clause to set specific properties for dependent tables when you issue the merge operation on the master table to merge partitions or subpartitions.

See Also: *Oracle Database SQL Language Reference*

If the involved partitions or subpartitions contain data, then indexes may be marked `UNUSABLE` as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: ■ The database marks `UNUSABLE` all resulting corresponding local index partitions or subpartitions. ■ Global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	■ The database marks `UNUSABLE` all resulting corresponding local index partitions. ■ All global indexes remain usable.

Merging Range Partitions

You are allowed to merge the contents of two adjacent range partitions into one partition. Nonadjacent range partitions cannot be merged. The resulting partition inherits the higher upper bound of the two merged partitions.

One reason for merging range partitions is to keep historical data online in larger partitions. For example, you can have daily partitions, with the oldest partition rolled up into weekly partitions, which can then be rolled up into monthly partitions, and so on.

The following scripts create an example of merging range partitions.

First, create a partitioned table and create local indexes.

```
-- Create a Table with four partitions each on its own tablespace
-- Partitioned by range on the data column.
--
CREATE TABLE four_seasons
(
        one DATE,
        two VARCHAR2(60),
        three NUMBER
)
PARTITION  BY RANGE ( one )
(
PARTITION quarter_one
   VALUES LESS THAN ( TO_DATE('01-apr-1998','dd-mon-yyyy'))
   TABLESPACE quarter_one,
PARTITION quarter_two
   VALUES LESS THAN ( TO_DATE('01-jul-1998','dd-mon-yyyy'))
   TABLESPACE quarter_two,
PARTITION quarter_three
   VALUES LESS THAN ( TO_DATE('01-oct-1998','dd-mon-yyyy'))
   TABLESPACE quarter_three,
PARTITION quarter_four
   VALUES LESS THAN ( TO_DATE('01-jan-1999','dd-mon-yyyy'))
   TABLESPACE quarter_four
);
--
-- Create local PREFIXED index on Four_Seasons
-- Prefixed because the leftmost columns of the index match the
-- Partitioning key
--
CREATE INDEX i_four_seasons_l ON four_seasons ( one,two )
LOCAL (
PARTITION i_quarter_one TABLESPACE i_quarter_one,
PARTITION i_quarter_two TABLESPACE i_quarter_two,
PARTITION i_quarter_three TABLESPACE i_quarter_three,
PARTITION i_quarter_four TABLESPACE i_quarter_four
);
```

Next, merge partitions.

```
--
-- Merge the first two partitions
--
ALTER TABLE four_seasons
MERGE PARTITIONS quarter_one, quarter_two INTO PARTITION quarter_two
UPDATE INDEXES;
```

If you omit the `UPDATE INDEXES` clause from the preceding statement, then you must rebuild the local index for the affected partition.

```
-- Rebuild index for quarter_two, which has been marked unusable
-- because it has not had all of the data from Q1 added to it.
-- Rebuilding the index will correct this.
--
ALTER TABLE four_seasons MODIFY PARTITION
quarter_two REBUILD UNUSABLE LOCAL INDEXES;
```

Merging Interval Partitions

The contents of two adjacent interval partitions can be merged into one partition. Nonadjacent interval partitions cannot be merged. The first interval partition can also be merged with the highest range partition. The resulting partition inherits the higher upper bound of the two merged partitions.

Merging interval partitions always results in the transition point being moved to the higher upper bound of the two merged partitions. This means that the range section of the interval-partitioned table will be extended to the upper bound of the two merged partitions. Any materialized interval partitions with boundaries lower than the newly merged partition will automatically be converted into range partitions, with their upper boundaries defined by the upper boundaries of their intervals.

For example, consider the following interval-partitioned table transactions:

```
CREATE TABLE transactions
( id               NUMBER
, transaction_date DATE
, value            NUMBER
)
PARTITION BY RANGE (transaction_date)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
( PARTITION p_before_2007 VALUES LESS THAN (TO_
DATE('01-JAN-2007','dd-MON-yyyy')));
```

Insert data into the interval section of the table. This will create the interval partitions for these days. Note that January 15, 2007 and January 16, 2007 are stored in adjacent interval partitions.

```
INSERT INTO transactions VALUES (1,TO_DATE('15-JAN-2007','dd-MON-yyyy'),100);
INSERT INTO transactions VALUES (2,TO_DATE('16-JAN-2007','dd-MON-yyyy'),600);
INSERT INTO transactions VALUES (3,TO_DATE('30-JAN-2007','dd-MON-yyyy'),200);
```

Next, merge the two adjacent interval partitions. The new partition will again have a system-generated name.

```
ALTER TABLE transactions
MERGE PARTITIONS FOR(TO_DATE('15-JAN-2007','dd-MON-yyyy'))
, FOR(TO_DATE('16-JAN-2007','dd-MON-yyyy'));
```

The transition point for the `transactions` table has now moved to January 17, 2007. The range section of the interval-partitioned table contains two range partitions: values less than January 1, 2007 and values less than January 17, 2007. Values greater than January 17, 2007 fall in the interval portion of the interval-partitioned table.

Merging List Partitions

When you merge list partitions, the partitions being merged can be any two partitions. They do not need to be adjacent, as for range partitions, because list partitioning does not assume any order for partitions. The resulting partition consists of all of the data from the original two partitions. If you merge a default list partition with any other partition, the resulting partition will be the default partition.

The following statement merges two partitions of a table partitioned using the list method into a partition that inherits all of its attributes from the table-level default attributes. `MAXEXTENTS` is specified in the statement.

```
ALTER TABLE q1_sales_by_region
   MERGE PARTITIONS q1_northcentral, q1_southcentral
   INTO PARTITION q1_central
      STORAGE(MAXEXTENTS 20);
```

The value lists for the two original partitions were specified as:

```
PARTITION q1_northcentral VALUES ('SD','WI')
PARTITION q1_southcentral VALUES ('OK','TX')
```

The resulting `sales_west` partition value list comprises the set that represents the union of these two partition value lists, or specifically:

```
('SD','WI','OK','TX')
```

Merging *-Hash Partitions

When you merge *-hash partitions, the subpartitions are rehashed into the number of subpartitions specified by `SUBPARTITIONS` *n* or the `SUBPARTITION` clause. If neither is included, table-level defaults are used.

Note that the inheritance of properties is different when a *-hash partition is split (discussed in "Splitting a *-Hash Partition" on page 3-61), as opposed to when two *-hash partitions are merged. When a partition is split, the new partitions can inherit properties from the original partition because there is only one parent. However, when partitions are merged, properties must be inherited from the table level.

For interval-hash partitioned tables, you can only merge two adjacent interval partitions, or the highest range partition with the first interval partition. As described in "Merging Interval Partitions" on page 3-47, the transition point will move when you merge intervals in an interval-hash partitioned table.

The following example merges two range-hash partitions:

```
ALTER TABLE all_seasons
   MERGE PARTITIONS quarter_1, quarter_2 INTO PARTITION quarter_2
   SUBPARTITIONS 8;
```

Merging *-List Partitions

Partitions can be merged at the partition level and subpartitions can be merged at the list subpartition level.

Merging Partitions in a *-List Partitioned Table Merging partitions in a *-list partitioned table is as described previously in "Merging Range Partitions" on page 3-46. However, when you merge two *-list partitions, the resulting new partition inherits the subpartition descriptions from the subpartition template, if one exists. If no subpartition template exists, then a single default subpartition is created for the new partition.

For interval-list partitioned tables, you can only merge two adjacent interval partitions, or the highest range partition with the first interval partition. As described in "Merging Interval Partitions" on page 3-47, the transition point will move when you merge intervals in an interval-list partitioned table.

The following statement merges two partitions in the range-list partitioned `stripe_regional_sales` table. A subpartition template exists for the table.

```
ALTER TABLE stripe_regional_sales
   MERGE PARTITIONS q1_1999, q2_1999 INTO PARTITION q1_q2_1999
      STORAGE(MAXEXTENTS 20);
```

Some new physical attributes are specified for this new partition while table-level defaults are inherited for those that are not specified. The new resulting partition `q1_q2_1999` inherits the high-value bound of the partition `q2_1999` and the subpartition value-list descriptions from the subpartition template description of the table.

The data in the resulting partitions consists of data from both the partitions. However, there may be cases where the database returns an error. This can occur because data may map out of the new partition when both of the following conditions exist:

- Some literal values of the merged subpartitions were not included in the subpartition template
- The subpartition template does not contain a default partition definition.

This error condition can be eliminated by always specifying a default partition in the default subpartition template.

Merging Subpartitions in a *-List Partitioned Table You can merge the contents of any two arbitrary list subpartitions belonging to the *same* partition. The resulting subpartition value-list descriptor includes all of the literal values in the value lists for the partitions being merged.

The following statement merges two subpartitions of a table partitioned using range-list method into a new subpartition located in tablespace `ts4`:

```
ALTER TABLE quarterly_regional_sales
   MERGE SUBPARTITIONS q1_1999_northwest, q1_1999_southwest
      INTO SUBPARTITION q1_1999_west
         TABLESPACE ts4;
```

The value lists for the original two partitions were:

- Subpartition `q1_1999_northwest` was described as `('WA','OR')`
- Subpartition `q1_1999_southwest` was described as `('AZ','NM','UT')`

The resulting subpartition value list comprises the set that represents the union of these two subpartition value lists:

- Subpartition `q1_1999_west` has a value list described as `('WA','OR','AZ','NM','UT')`

The tablespace in which the resulting subpartition is located and the subpartition attributes are determined by the partition-level default attributes, except for those specified explicitly. If any of the existing subpartition names are being reused, then the new subpartition inherits the subpartition attributes of the subpartition whose name is being reused.

Merging *-Range Partitions

Partitions can be merged at the partition level and subpartitions can be merged at the range subpartition level.

Merging Partitions in a *-Range Partitioned Table Merging partitions in a *-range partitioned table is as described previously in "Merging Range Partitions" on page 3-46. However, when you merge two *-range partitions, the resulting new partition inherits the subpartition descriptions from the subpartition template, if one exists. If no subpartition template exists, then a single subpartition with an upper boundary `MAXVALUE` is created for the new partition.

For interval-range partitioned tables, you can only merge two adjacent interval partitions, or the highest range partition with the first interval partition. As described in "Merging Interval Partitions" on page 3-47, the transition point will move when you merge intervals in an interval-range partitioned table.

The following statement merges two partitions in the monthly interval-range partitioned `orders` table. A subpartition template exists for the table.

```
ALTER TABLE orders
MERGE PARTITIONS FOR(TO_DATE('01-MAR-2007','dd-MON-yyyy')),
FOR(TO_DATE('01-APR-2007','dd-MON-yyyy'))
INTO PARTITION p_pre_may_2007;
```

If the March 2007 and April 2007 partitions were still in the interval section of the interval-range partitioned table, then the merge operation would move the transition point to May 1, 2007.

The subpartitions for partition `p_pre_may_2007` inherit their properties from the subpartition template. The data in the resulting partitions consists of data from both the partitions. However, there may be cases where the database returns an error. This can occur because data may map out of the new partition when both of the following conditions are met:

- Some range values of the merged subpartitions were not included in the subpartition template.
- The subpartition template does not have a subpartition definition with a `MAXVALUE` upper boundary.

The error condition can be eliminated by always specifying a subpartition with an upper boundary of `MAXVALUE` in the subpartition template.

Modifying Default Attributes

You can modify the default attributes of a table, or for a partition of a composite partitioned table. When you modify default attributes, the new attributes affect only future partitions, or subpartitions, that are created. The default values can still be specifically overridden when creating a new partition or subpartition. You can modify the default attributes of a reference-partitioned table.

Modifying Default Attributes of a Table

You can modify the default attributes that will be inherited for range, hash, list, interval, or reference partitions using the `MODIFY DEFAULT ATTRIBUTES` clause of `ALTER TABLE`.

For hash-partitioned tables, only the `TABLESPACE` attribute can be modified.

Modifying Default Attributes of a Partition

To modify the default attributes inherited when creating subpartitions, use the `ALTER TABLE` ... `MODIFY DEFAULT ATTRIBUTES FOR PARTITION`. The following statement modifies the `TABLESPACE` in which future subpartitions of partition `p1` in range-hash partitioned table `emp` will reside.

```
ALTER TABLE emp
     MODIFY DEFAULT ATTRIBUTES FOR PARTITION p1 TABLESPACE ts1;
```

Because all subpartitions of a range-hash partitioned table must share the same attributes, except `TABLESPACE`, it is the only attribute that can be changed.

You cannot modify default attributes of interval partitions that have not yet been created. If you want to change the way in which future subpartitions in an interval-partitioned table are created, then you have to modify the subpartition template.

Modifying Default Attributes of Index Partitions

In similar fashion to table partitions, you can alter the default attributes that will be inherited by partitions of a range-partitioned global index, or local index partitions of partitioned tables. For this you use the `ALTER INDEX` ... `MODIFY DEFAULT ATTRIBUTES` statement. Use the `ALTER INDEX` ... `MODIFY DEFAULT ATTRIBUTES FOR PARTITION` statement if you are altering default attributes to be inherited by subpartitions of a composite partitioned table.

Modifying Real Attributes of Partitions

It is possible to modify attributes of an existing partition of a table or index.

You cannot change the `TABLESPACE` attribute. Use `ALTER TABLESPACE` ... `MOVE PARTITION/SUBPARTITION` to move a partition or subpartition to a new tablespace.

Modifying Real Attributes for a Range or List Partition

Use the `ALTER TABLE` ... `MODIFY PARTITION` statement to modify existing attributes of a range partition or list partition. You can modify segment attributes (except `TABLESPACE`), or you can allocate and deallocate extents, mark local index partitions `UNUSABLE`, or rebuild local indexes that have been marked `UNUSABLE`.

If this is a range partition of a *-hash partitioned table, then note the following:

- If you allocate or deallocate an extent, this action is performed for every subpartition of the specified partition.

- Likewise, changing any other attributes results in corresponding changes to those attributes of all the subpartitions for that partition. The partition level default attributes are changed as well. To avoid changing attributes of existing subpartitions, use the `FOR PARTITION` clause of the `MODIFY DEFAULT ATTRIBUTES` statement.

The following are some examples of modifying the real attributes of a partition.

This example modifies the `MAXEXTENTS` storage attribute for the range partition `sales_q1` of table `sales`:

```
ALTER TABLE sales MODIFY PARTITION sales_q1
     STORAGE (MAXEXTENTS 10);
```

All of the local index subpartitions of partition `ts1` in range-hash partitioned table scubagear are marked `UNUSABLE` in the following example:

```
ALTER TABLE scubagear MODIFY PARTITION ts1 UNUSABLE LOCAL INDEXES;
```

For an interval-partitioned table you can only modify real attributes of range partitions or interval partitions that have been created.

Modifying Real Attributes for a Hash Partition

You also use the `ALTER TABLE` ... `MODIFY PARTITION` statement to modify attributes of a hash partition. However, because the physical attributes of individual hash partitions must all be the same (except for `TABLESPACE`), you are restricted to:

- Allocating a new extent
- Deallocating an unused extent
- Marking a local index subpartition `UNUSABLE`
- Rebuilding local index subpartitions that are marked `UNUSABLE`

The following example rebuilds any unusable local index partitions associated with hash partition `P1` of table `dept`:

```
ALTER TABLE dept MODIFY PARTITION p1
     REBUILD UNUSABLE LOCAL INDEXES;
```

Modifying Real Attributes of a Subpartition

With the `MODIFY SUBPARTITION` clause of `ALTER TABLE` you can perform the same actions as listed previously for partitions, but at the specific composite partitioned table subpartition level. For example:

```
ALTER TABLE emp MODIFY SUBPARTITION p3_s1
     REBUILD UNUSABLE LOCAL INDEXES;
```

Modifying Real Attributes of Index Partitions

The `MODIFY PARTITION` clause of `ALTER INDEX` lets you modify the real attributes of an index partition or its subpartitions. The rules are very similar to those for table partitions, but unlike the `MODIFY PARTITION` clause for `ALTER INDEX`, there is no subclause to rebuild an unusable index partition, but there is a subclause to coalesce an index partition or its subpartitions. In this context, coalesce means to merge index blocks where possible to free them for reuse.

You can also allocate or deallocate storage for a subpartition of a local index, or mark it `UNUSABLE`, using the `MODIFY PARTITION` clause.

Modifying List Partitions: Adding Values

List partitioning allows you the option of adding literal values from the defining value list.

Adding Values for a List Partition

Use the MODIFY PARTITION ... ADD VALUES clause of the ALTER TABLE statement to extend the value list of an existing partition. Literal values being added must not have been included in any other partition value list. The partition value list for any corresponding local index partition is correspondingly extended, and any global indexes, or global or local index partitions, remain usable.

The following statement adds a new set of state codes ('OK', 'KS') to an existing partition list.

```
ALTER TABLE sales_by_region
   MODIFY PARTITION region_south
      ADD VALUES ('OK', 'KS');
```

The existence of a default partition can have a performance impact when adding values to other partitions. This is because in order to add values to a list partition, the database must check that the values being added do not already exist in the default partition. If any of the values do exist in the default partition, then an error is raised.

Note: The database executes a query to check for the existence of rows in the default partition that correspond to the literal values being added. Therefore, it is advisable to create a local prefixed index on the table. This speeds up the execution of the query and the overall operation.

You cannot add values to a default list partition.

Adding Values for a List Subpartition

This operation is essentially the same as described for "Modifying List Partitions: Adding Values", however, you use a MODIFY SUBPARTITION clause instead of the MODIFY PARTITION clause. For example, to extend the range of literal values in the value list for subpartition q1_1999_southeast, use the following statement:

```
ALTER TABLE quarterly_regional_sales
   MODIFY SUBPARTITION q1_1999_southeast
      ADD VALUES ('KS');
```

Literal values being added must not have been included in any other subpartition value list within the owning partition. However, they can be duplicates of literal values in the subpartition value lists of other partitions within the table.

For an interval-list composite partitioned table, you can only add values to subpartitions of range partitions or interval partitions that have been created. If you want to add values to subpartitions of interval partitions that have not yet been created, then you have to modify the subpartition template.

Modifying List Partitions: Dropping Values

List partitioning allows you the option of dropping literal values from the defining value list.

Dropping Values from a List Partition

Use the MODIFY PARTITION ... DROP VALUES clause of the ALTER TABLE statement to remove literal values from the value list of an existing partition. The statement is always executed with validation, meaning that it checks to see if any rows exist in the partition that corresponds to the set of values being dropped. If any such rows are found then the database returns an error message and the operation fails. When necessary, use a DELETE statement to delete corresponding rows from the table before attempting to drop values.

Note: You cannot drop all literal values from the value list describing the partition. You must use the ALTER TABLE ... DROP PARTITION statement instead.

The partition value list for any corresponding local index partition reflects the new value list, and any global index, or global or local index partitions, remain usable.

The following statement drops a set of state codes ('OK' and 'KS') from an existing partition value list.

```
ALTER TABLE sales_by_region
   MODIFY PARTITION region_south
      DROP VALUES ('OK', 'KS');
```

Note: The database executes a query to check for the existence of rows in the partition that correspond to the literal values being dropped. Therefore, it is advisable to create a local prefixed index on the table. This speeds up the execution of the query and the overall operation.

You cannot drop values from a default list partition.

Dropping Values from a List Subpartition

This operation is essentially the same as described for "Modifying List Partitions: Dropping Values", however, you use a MODIFY SUBPARTITION clause instead of the MODIFY PARTITION clause. For example, to remove a set of literal values in the value list for subpartition q1_1999_southeast, use the following statement:

```
ALTER TABLE quarterly_regional_sales
   MODIFY SUBPARTITION q1_1999_southeast
      DROP VALUES ('KS');
```

For an interval-list composite partitioned table, you can only drop values from subpartitions of range partitions or interval partitions that have been created. If you want to drop values from subpartitions of interval partitions that have not yet been created, then you have to modify the subpartition template.

Modifying a Subpartition Template

You can modify a subpartition template of a composite partitioned table by replacing it with a new subpartition template. Any subsequent operations that use the subpartition template (such as ADD PARTITION or MERGE PARTITIONS) will now use the new subpartition template. Existing subpartitions remain unchanged.

If you modify a subpartition template of an interval-* composite partitioned table, then interval partitions that have not yet been created will use the new subpartition template.

Use the `ALTER TABLE` ... `SET SUBPARTITION TEMPLATE` statement to specify a new subpartition template. For example:

```
ALTER TABLE emp_sub_template
   SET SUBPARTITION TEMPLATE
         (SUBPARTITION e TABLESPACE ts1,
          SUBPARTITION f TABLESPACE ts2,
          SUBPARTITION g TABLESPACE ts3,
          SUBPARTITION h TABLESPACE ts4
         );
```

You can drop a subpartition template by specifying an empty list:

```
ALTER TABLE emp_sub_template
   SET SUBPARTITION TEMPLATE ( );
```

Moving Partitions

Use the `MOVE PARTITION` clause of the `ALTER TABLE` statement to:

- Re-cluster data and reduce fragmentation
- Move a partition to another tablespace
- Modify create-time attributes
- Store the data in compressed format using table compression

Typically, you can change the physical storage attributes of a partition in a single step using an `ALTER TABLE/INDEX` ... `MODIFY PARTITION` statement. However, there are some physical attributes, such as `TABLESPACE`, that you cannot modify using `MODIFY PARTITION`. In these cases, use the `MOVE PARTITION` clause. Modifying some other attributes, such as table compression, affects only future storage, but not existing data.

Note: `ALTER TABLE`...`MOVE` does not permit DML on the partition while the command is executing. If you want to move a partition and leave it available for DML, see "Redefining Partitions Online" on page 3-56.

If the partition being moved contains any data, indexes may be marked `UNUSABLE` according to the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: ■ The matching partition in each local index is marked `UNUSABLE`. You must rebuild these index partitions after issuing `MOVE PARTITION`. ■ Any global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE`.
Index-organized	Any local or global indexes defined for the partition being moved remain usable because they are primary-key based logical rowids. However, the guess information for these rowids becomes incorrect.

Moving Table Partitions

Use the MOVE PARTITION clause to move a partition. For example, to move the most active partition to a tablespace that resides on its own set of disks (in order to balance I/O), not log the action, and compress the data, issue the following statement:

```
ALTER TABLE parts MOVE PARTITION depot2
     TABLESPACE ts094 NOLOGGING COMPRESS;
```

This statement always drops the old partition segment and creates a new segment, even if you do not specify a new tablespace.

If you are moving a partition of a partitioned index-organized table, then you can specify the MAPPING TABLE clause as part of the MOVE PARTITION clause, and the mapping table partition will be moved to the new location along with the table partition.

For an interval or interval-* partitioned table, you can only move range partitions or interval partitions that have been created. A partition move operation does not move the transition point in an interval or interval-* partitioned table.

You can move a partition in a reference-partitioned table independent of the partition in the master table.

Moving Subpartitions

The following statement shows how to move data in a subpartition of a table. In this example, a PARALLEL clause has also been specified.

```
ALTER TABLE scuba_gear MOVE SUBPARTITION bcd_types
     TABLESPACE tbs23 PARALLEL (DEGREE 2);
```

You can move a subpartition in a reference-partitioned table independent of the subpartition in the master table.

Moving Index Partitions

The ALTER TABLE ... MOVE PARTITION statement for regular tables, marks all partitions of a global index UNUSABLE. You can rebuild the entire index by rebuilding each partition individually using the ALTER INDEX ... REBUILD PARTITION statement. You can perform these rebuilds concurrently.

You can also simply drop the index and re-create it.

Redefining Partitions Online

Oracle Database provides a mechanism to move a partition or to make other changes to the partition's physical structure without significantly affecting the availability of the partition for DML. The mechanism is called **online table redefinition**.

For information on redefining a single partition of a table, see *Oracle Database Administrator's Guide*.

Rebuilding Index Partitions

Some reasons for rebuilding index partitions include:

- To recover space and improve performance
- To repair a damaged index partition caused by media failure
- To rebuild a local index partition after loading the underlying table partition with SQL*Loader or an import utility

- To rebuild index partitions that have been marked `UNUSABLE`
- To enable key compression for B-tree indexes

The following sections discuss options for rebuilding index partitions and subpartitions.

Rebuilding Global Index Partitions

You can rebuild global index partitions in two ways:

- Rebuild each partition by issuing the `ALTER INDEX` ... `REBUILD PARTITION` statement (you can run the rebuilds concurrently).
- Drop the entire global index and re-create it. This method is more efficient because the table is scanned only once.

For most maintenance operations on partitioned tables with indexes, you can optionally avoid the need to rebuild the index by specifying `UPDATE INDEXES` on your DDL statement.

Rebuilding Local Index Partitions

Rebuild local indexes using either `ALTER INDEX` or `ALTER TABLE` as follows:

- `ALTER INDEX` ... `REBUILD PARTITION/SUBPARTITION`

  This statement rebuilds an index partition or subpartition unconditionally.
- `ALTER TABLE` ... `MODIFY PARTITION/SUBPARTITION` ... `REBUILD UNUSABLE LOCAL INDEXES`

  This statement finds all of the unusable indexes for the given table partition or subpartition and rebuilds them. It only rebuilds an index partition if it has been marked `UNUSABLE`.

Using ALTER INDEX to Rebuild a Partition The `ALTER INDEX` ... `REBUILD PARTITION` statement rebuilds one partition of an index. It cannot be used for composite-partitioned tables. Only real physical segments can be rebuilt with this command. When you re-create the index, you can also choose to move the partition to a new tablespace or change attributes.

For composite-partitioned tables, use `ALTER INDEX` ... `REBUILD SUBPARTITION` to rebuild a subpartition of an index. You can move the subpartition to another tablespace or specify a parallel clause. The following statement rebuilds a subpartition of a local index on a table and moves the index subpartition is another tablespace.

```
ALTER INDEX scuba
   REBUILD SUBPARTITION bcd_types
   TABLESPACE tbs23 PARALLEL (DEGREE 2);
```

Using ALTER TABLE to Rebuild an Index Partition The `REBUILD UNUSABLE LOCAL INDEXES` clause of `ALTER TABLE` ... `MODIFY PARTITION` does not allow you to specify any new attributes for the rebuilt index partition. The following example finds and rebuilds any unusable local index partitions for table `scubagear`, partition `p1`.

```
ALTER TABLE scubagear
   MODIFY PARTITION p1 REBUILD UNUSABLE LOCAL INDEXES;
```

There is a corresponding `ALTER TABLE` ... `MODIFY SUBPARTITION` clause for rebuilding unusable local index subpartitions.

Renaming Partitions

It is possible to rename partitions and subpartitions of both tables and indexes. One reason for renaming a partition might be to assign a meaningful name, as opposed to a default system name that was assigned to the partition in another maintenance operation.

All partitioning methods support the `FOR(`*`value`*`)` method to identify a partition. You can use this method to rename a system-generated partition name into a more meaningful name. This is particularly useful in interval or interval-* partitioned tables.

You can independently rename partitions and subpartitions for reference-partitioned master and child tables. The rename operation on the master table is not cascaded to descendant tables.

Renaming a Table Partition

Rename a range, hash, or list partition, using the `ALTER TABLE` ... `RENAME PARTITION` statement. For example:

```
ALTER TABLE scubagear RENAME PARTITION sys_p636 TO tanks;
```

Renaming a Table Subpartition

Likewise, you can assign new names to subpartitions of a table. In this case you would use the `ALTER TABLE` ... `RENAME SUBPARTITION` syntax.

Renaming Index Partitions

Index partitions and subpartitions can be renamed in similar fashion, but the `ALTER INDEX` syntax is used.

Renaming an Index Partition Use the `ALTER INDEX` ... `RENAME PARTITION` statement to rename an index partition.

The `ALTER  INDEX` statement does not support the use of `FOR(`*`value`*`)` to identify a partition. You have to use the original partition name in the rename operation.

Renaming an Index Subpartition This next statement simply shows how to rename a subpartition that has a system generated name that was a consequence of adding a partition to an underlying table:

```
ALTER INDEX scuba RENAME SUBPARTITION sys_subp3254 TO bcd_types;
```

Splitting Partitions

The `SPLIT PARTITION` clause of the `ALTER TABLE` or `ALTER INDEX` statement is used to redistribute the contents of a partition into two new partitions. Consider doing this when a partition becomes too large and causes backup, recovery, or maintenance operations to take a long time to complete or it is felt that there is simply too much data in the partition. You can also use the `SPLIT PARTITION` clause to redistribute the I/O load.

This clause cannot be used for hash partitions or subpartitions.

If the partition you are splitting contains any data, then indexes may be marked `UNUSABLE` as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: ▪ The database marks `UNUSABLE` the new partitions (there are two) in each local index. ▪ Any global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	▪ The database marks `UNUSABLE` the new partitions (there are two) in each local index. ▪ All global indexes remain usable.

You cannot split partitions or subpartitions in a reference-partitioned table. When you split partitions or subpartitions in the parent table then the split is cascaded to all descendant tables. However, you can use the `DEPENDENT TABLES` clause to set specific properties for dependent tables when you issue the `SPLIT` statement on the master table to split partitions or subpartitions.

See Also: *Oracle Database SQL Language Reference*

Splitting a Partition of a Range-Partitioned Table

You split a range partition using the `ALTER TABLE ... SPLIT PARTITION` statement. You specify a value of the partitioning key column within the range of the partition at which to split the partition. The first of the resulting two new partitions includes all rows in the original partition whose partitioning key column values map lower that the specified value. The second partition contains all rows whose partitioning key column values map greater than or equal to the specified value.

You can optionally specify new attributes for the two partitions resulting from the split. If there are local indexes defined on the table, this statement also splits the matching partition in each local index.

In the following example `fee_katy` is a partition in the table `vet_cats`, which has a local index, `jaf1`. There is also a global index, `vet` on the table. `vet` contains two partitions, `vet_parta`, and `vet_partb`.

To split the partition `fee_katy`, and rebuild the index partitions, issue the following statements:

```
ALTER TABLE vet_cats SPLIT PARTITION
      fee_katy at (100) INTO ( PARTITION
      fee_katy1, PARTITION fee_katy2);
ALTER INDEX JAF1 REBUILD PARTITION fee_katy1;
ALTER INDEX JAF1 REBUILD PARTITION fee_katy2;
ALTER INDEX VET REBUILD PARTITION vet_parta;
ALTER INDEX VET REBUILD PARTITION vet_partb;
```

Note: If you do not specify new partition names, the database assigns names of the form `SYS_P`*n*. You can examine the data dictionary to locate the names assigned to the new local index partitions. You may want to rename them. Any attributes you do not specify are inherited from the original partition.

Splitting a Partition of a List-Partitioned Table

You split a list partition by using the `ALTER TABLE` ... `SPLIT PARTITION` statement. The `SPLIT PARTITION` clause enables you to specify a list of literal values that define a partition into which rows with corresponding partitioning key values are inserted. The remaining rows of the original partition are inserted into a second partition whose value list contains the remaining values from the original partition.

You can optionally specify new attributes for the two partitions that result from the split.

The following statement splits the partition `region_east` into two partitions:

```
ALTER TABLE sales_by_region
   SPLIT PARTITION region_east VALUES ('CT', 'MA', 'MD')
   INTO
    ( PARTITION region_east_1
         TABLESPACE tbs2,
      PARTITION region_east_2
        STORAGE (NEXT 2M PCTINCREASE 25))
   PARALLEL 5;
```

The literal value list for the original `region_east` partition was specified as:

```
PARTITION region_east VALUES ('MA','NY','CT','NH','ME','MD','VA','PA','NJ')
```

The two new partitions are:

- `region_east_1` with a literal value list of `('CT','MA','MD')`
- `region_east_2` inheriting the remaining literal value list of `('NY','NH','ME','VA','PA','NJ')`

The individual partitions have new physical attributes specified at the partition level. The operation is executed with parallelism of degree 5.

You can split a default list partition just like you split any other list partition. This is also the only means of adding a partition to a list-partitioned table that contains a default partition. When you split the default partition, you create a new partition defined by the values that you specify, and a second partition that remains the default partition.

The following example splits the default partition of `sales_by_region`, thereby creating a new partition:

```
ALTER TABLE sales_by_region
   SPLIT PARTITION region_unknown VALUES ('MT', 'WY', 'ID')
   INTO
    ( PARTITION region_wildwest,
      PARTITION region_unknown);
```

Splitting a Partition of an Interval-Partitioned Table

You split a range or a materialized interval partition using the `ALTER TABLE ... SPLIT PARTITION` statement in an interval-partitioned table. Splitting a range partition in the interval-partitioned table is the same as described in "Splitting a Partition of a Range-Partitioned Table" on page 3-59.

To split a materialized interval partition, you specify a value of the partitioning key column within the interval partition at which to split the partition. The first of the resulting two new partitions includes all rows in the original partition whose partitioning key column values map lower than the specified value. The second partition contains all rows whose partitioning key column values map greater than or equal to the specified value. The split partition operation will move the transition point up to the higher boundary of the partition you just split, and all materialized interval partitions lower than the newly split partitions are implicitly converted into range partitions, with their upper boundaries defined by the upper boundaries of the intervals.

You can optionally specify new attributes for the two range partitions resulting from the split. If there are local indexes defined on the table, then this statement also splits the matching partition in each local index. You cannot split interval partitions that have not yet been created.

The following example shows splitting the May 2007 partition in the monthly interval partitioned table `transactions`.

```
ALTER TABLE transactions
SPLIT PARTITION FOR(TO_DATE('01-MAY-2007','dd-MON-yyyy'))
AT (TO_DATE('15-MAY-2007','dd-MON-yyyy'));
```

Splitting a *-Hash Partition

This is the opposite of merging *-hash partitions. When you split *-hash partitions, the new subpartitions are rehashed into either the number of subpartitions specified in a `SUBPARTITIONS` or `SUBPARTITION` clause. Or, if no such clause is included, the new partitions inherit the number of subpartitions (and tablespaces) from the partition being split.

Note that the inheritance of properties is different when a *-hash partition is split, versus when two *-hash partitions are merged. When a partition is split, the new partitions can inherit properties from the original partition because there is only one parent. However, when partitions are merged, properties must be inherited from *table level* defaults because there are two parents and the new partition cannot inherit from either at the expense of the other.

The following example splits a range-hash partition:

```
ALTER TABLE all_seasons SPLIT PARTITION quarter_1
     AT (TO_DATE('16-dec-1997','dd-mon-yyyy'))
     INTO (PARTITION q1_1997_1 SUBPARTITIONS 4 STORE IN (ts1,ts3),
           PARTITION q1_1997_2);
```

The rules for splitting an interval-hash partitioned table follow the rules for splitting an interval-partitioned table. As described in "Splitting a Partition of an Interval-Partitioned Table" on page 3-60, the transition point will be changed to the higher boundary of the split partition.

Splitting Partitions in a *-List Partitioned Table

Partitions can be split at both the partition level and at the list subpartition level.

Splitting a *-List Partition Splitting a partition of a *-list partitioned table is similar to what is described in "Splitting a Partition of a Range-Partitioned Table" on page 3-59. No subpartition literal value list can be specified for either of the new partitions. The new partitions inherit the subpartition descriptions from the original partition being split.

The following example splits the `q1_1999` partition of the `quarterly_regional_sales` table:

```
ALTER TABLE quarterly_regional_sales SPLIT PARTITION q1_1999
```

```
   AT (TO_DATE('15-Feb-1999','dd-mon-yyyy'))
   INTO ( PARTITION q1_1999_jan_feb
             TABLESPACE ts1,
          PARTITION q1_1999_feb_mar
             STORAGE (NEXT 2M PCTINCREASE 25) TABLESPACE ts2)
   PARALLEL 5;
```

This operation splits the partition `q1_1999` into two resulting partitions: `q1_1999_jan_feb` and `q1_1999_feb_mar`. Both partitions inherit their subpartition descriptions from the original partition. The individual partitions have new physical attributes, including tablespaces, specified at the partition level. These new attributes become the default attributes of the new partitions. This operation is run with parallelism of degree 5.

The `ALTER TABLE` ... `SPLIT PARTITION` statement provides no means of specifically naming subpartitions resulting from the split of a partition in a composite partitioned table. However, for those subpartitions in the parent partition with names of the form *`partition name_subpartition name`*, the database generates corresponding names in the newly created subpartitions using the new partition names. All other subpartitions are assigned system generated names of the form `SYS_SUBP`*n*. System generated names are also assigned for the subpartitions of any partition resulting from the split for which a name is not specified. Unnamed partitions are assigned a system generated partition name of the form `SYS_P`*n*.

The following query displays the subpartition names resulting from the previous split partition operation on table `quarterly_regional_sales`. It also reflects the results of other operations performed on this table in preceding sections of this chapter since its creation in "Creating Composite Range-List Partitioned Tables" on page 3-9.

```
SELECT PARTITION_NAME, SUBPARTITION_NAME, TABLESPACE_NAME
  FROM DBA_TAB_SUBPARTITIONS
  WHERE TABLE_NAME='QUARTERLY_REGIONAL_SALES'
  ORDER BY PARTITION_NAME;

PARTITION_NAME       SUBPARTITION_NAME              TABLESPACE_NAME
-------------------- ------------------------------ ---------------
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_WEST           TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_NORTHEAST      TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_SOUTHEAST      TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_NORTHCENTRAL   TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_SOUTHCENTRAL   TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_SOUTH          TS2
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_WEST           TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_NORTHEAST      TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_SOUTHEAST      TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_NORTHCENTRAL   TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_SOUTHCENTRAL   TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_SOUTH          TS1
Q1_2000              Q1_2000_NORTHWEST              TS3
Q1_2000              Q1_2000_SOUTHWEST              TS3
Q1_2000              Q1_2000_NORTHEAST              TS3
Q1_2000              Q1_2000_SOUTHEAST              TS3
Q1_2000              Q1_2000_NORTHCENTRAL           TS3
Q1_2000              Q1_2000_SOUTHCENTRAL           TS3
Q2_1999              Q2_1999_NORTHWEST              TS4
Q2_1999              Q2_1999_SOUTHWEST              TS4
Q2_1999              Q2_1999_NORTHEAST              TS4
Q2_1999              Q2_1999_SOUTHEAST              TS4
Q2_1999              Q2_1999_NORTHCENTRAL           TS4
Q2_1999              Q2_1999_SOUTHCENTRAL           TS4
```

```
Q3_1999             Q3_1999_NORTHWEST              TS4
Q3_1999             Q3_1999_SOUTHWEST              TS4
Q3_1999             Q3_1999_NORTHEAST              TS4
Q3_1999             Q3_1999_SOUTHEAST              TS4
Q3_1999             Q3_1999_NORTHCENTRAL           TS4
Q3_1999             Q3_1999_SOUTHCENTRAL           TS4
Q4_1999             Q4_1999_NORTHWEST              TS4
Q4_1999             Q4_1999_SOUTHWEST              TS4
Q4_1999             Q4_1999_NORTHEAST              TS4
Q4_1999             Q4_1999_SOUTHEAST              TS4
Q4_1999             Q4_1999_NORTHCENTRAL           TS4
Q4_1999             Q4_1999_SOUTHCENTRAL           TS4

36 rows selected.
```

Splitting a *-List Subpartition Splitting a list subpartition of a *-list partitioned table is similar to what is described in "Splitting a Partition of a List-Partitioned Table" on page 3-60, but the syntax is that of `SUBPARTITION` rather than `PARTITION`. For example, the following statement splits a subpartition of the `quarterly_regional_sales` table:

```
ALTER TABLE quarterly_regional_sales SPLIT SUBPARTITION q2_1999_southwest
   VALUES ('UT') INTO
      ( SUBPARTITION q2_1999_utah
           TABLESPACE ts2,
        SUBPARTITION q2_1999_southwest
           TABLESPACE ts3
      )
   PARALLEL;
```

This operation splits the subpartition `q2_1999_southwest` into two subpartitions:

- `q2_1999_utah` with literal value list of `('UT')`
- `q2_1999_southwest` which inherits the remaining literal value list of `('AZ','NM')`

The individual subpartitions have new physical attributes that are inherited from the subpartition being split.

You can only split subpartitions in an interval-list partitioned table for range partitions or materialized interval partitions. If you want to change subpartition values for future interval partitions, then you have to modify the subpartition template.

Splitting a *-Range Partition

Splitting a partition of a *-range partitioned table is similar to what is described in "Splitting a Partition of a Range-Partitioned Table" on page 3-59. No subpartition range values can be specified for either of the new partitions. The new partitions inherit the subpartition descriptions from the original partition being split.

The following example splits the May 2007 interval partition of the interval-range partitioned `orders` table:

```
ALTER TABLE orders
SPLIT PARTITION FOR(TO_DATE('01-MAY-2007','dd-MON-yyyy'))
AT (TO_DATE('15-MAY-2007','dd-MON-yyyy'))
INTO (PARTITION p_fh_may07,PARTITION p_sh_may2007);
```

This operation splits the interval partition `FOR('01-MAY-2007')` into two resulting partitions: `p_fh_may07` and `p_sh_may_2007`. Both partitions inherit their subpartition descriptions from the original partition. Any interval partitions before the June 2007 partition have been converted into range partitions, as described in "Merging Interval Partitions" on page 3-47.

The `ALTER TABLE ... SPLIT PARTITION` statement provides no means of specifically naming subpartitions resulting from the split of a partition in a composite partitioned table. However, for those subpartitions in the parent partition with names of the form *`partition name_subpartition name`*, the database generates corresponding names in the newly created subpartitions using the new partition names. All other subpartitions are assigned system generated names of the form `SYS_SUBP`*`n`*. System generated names are also assigned for the subpartitions of any partition resulting from the split for which a name is not specified. Unnamed partitions are assigned a system generated partition name of the form `SYS_P`*`n`*.

The following query displays the subpartition names and high values resulting from the previous split partition operation on table `orders`. It also reflects the results of other operations performed on this table in preceding sections of this chapter since its creation.

```
BREAK ON partition_name

SELECT partition_name, subpartition_name, high_value
FROM user_tab_subpartitions
WHERE table_name = 'ORCERS'
ORDER BY partition_name, subpartition_position;

PARTITION_NAME            SUBPARTITION_NAME              HIGH_VALUE
------------------------- ------------------------------ ---------------
P_BEFORE_2007             P_BEFORE_2007_P_SMALL          1000
                          P_BEFORE_2007_P_MEDIUM         10000
                          P_BEFORE_2007_P_LARGE          100000
                          P_BEFORE_2007_P_EXTRAORDINARY  MAXVALUE
P_FH_MAY07                SYS_SUBP2985                   1000
                          SYS_SUBP2986                   10000
                          SYS_SUBP2987                   100000
                          SYS_SUBP2988                   MAXVALUE
P_PRE_MAY_2007            P_PRE_MAY_2007_P_SMALL         1000
                          P_PRE_MAY_2007_P_MEDIUM        10000
                          P_PRE_MAY_2007_P_LARGE         100000
                          P_PRE_MAY_2007_P_EXTRAORDINARY MAXVALUE
P_SH_MAY2007              SYS_SUBP2989                   1000
                          SYS_SUBP2990                   10000
                          SYS_SUBP2991                   100000
                          SYS_SUBP2992                   MAXVALUE
```

Splitting a *-Range Subpartition Splitting a range subpartition of a *-range partitioned table is similar to what is described in "Splitting a Partition of a Range-Partitioned Table" on page 3-59, but the syntax is that of SUBPARTITION rather than PARTITION. For example, the following statement splits a subpartition of the `orders` table:

```
ALTER TABLE orders
SPLIT SUBPARTITION p_pre_may_2007_p_large AT (50000)
INTO (SUBPARTITION p_pre_may_2007_med_large TABLESPACE TS4
     , SUBPARTITION p_pre_may_2007_large_large TABLESPACE TS5
     );
```

This operation splits the subpartition `p_pre_may_2007_p_large` into two subpartitions:

- `p_pre_may_2007_med_large` with values between 10000 and 50000
- `p_pre_may_2007_large_large` with values between 50000 and 100000

The individual subpartitions have new physical attributes that are inherited from the subpartition being split.

You can only split subpartitions in an interval-range partitioned table for range partitions or materialized interval partitions. If you want to change subpartition boundaries for future interval partitions, then you have to modify the subpartition template.

Splitting Index Partitions

You cannot explicitly split a partition in a local index. A local index partition is split only when you split a partition in the underlying table. However, you can split a global index partition as is done in the following example:

```
ALTER INDEX quon1 SPLIT
    PARTITION canada AT ( 100 ) INTO
    PARTITION canada1 ..., PARTITION canada2 ...);
ALTER INDEX quon1 REBUILD PARTITION canada1;
ALTER INDEX quon1 REBUILD PARTITION canada2;
```

The index being split can contain index data, and the resulting partitions do not require rebuilding, unless the original partition was previously marked `UNUSABLE`.

Optimizing SPLIT PARTITION and SPLIT SUBPARTITION Operations

Oracle Database implements a `SPLIT PARTITION` operation by creating two new partitions and redistributing the rows from the partition being split into the two new partitions. This is an expensive operation because it is necessary to scan all the rows of the partition being split and then insert them one-by-one into the new partitions. Further if you do not use the `UPDATE INDEXES` clause, both local and global indexes also require rebuilding.

Sometimes after a split operation, one of the new partitions contains all of the rows from the partition being split, while the other partition contains no rows. This is often the case when splitting the first partition of a table. The database can detect such situations and can optimize the split operation. This optimization results in a fast split operation that behaves like an add partition operation.

Specifically, the database can optimize and speed up `SPLIT PARTITION` operations if all of the following conditions are met:

- One of the two resulting partitions must be empty.
- The non-empty resulting partition must have storage characteristics identical to those of the partition being split. Specifically:
  - If the partition being split is composite, then the storage characteristics of each subpartition in the new non-empty resulting partition must be identical to those of the subpartitions of the partition being split.
  - If the partition being split contains a `LOB` column, then the storage characteristics of each `LOB` (sub)partition in the new non-empty resulting partition must be identical to those of the `LOB` (sub)partitions of the partition being split.
  - If a partition of an index-organized table with overflow is being split, then the storage characteristics of each overflow (sub)partition in the new nonempty

resulting partition must be identical to those of the overflow (sub)partitions of the partition being split.

– If a partition of an index-organized table with mapping table is being split, then the storage characteristics of each mapping table (sub)partition in the new nonempty resulting partition must be identical to those of the mapping table (sub)partitions of the partition being split.

If these conditions are met after the split, then all global indexes remain usable, even if you did not specify the `UPDATE INDEXES` clause. Local index (sub)partitions associated with both resulting partitions remain usable if they were usable before the split. Local index (sub)partition(s) corresponding to the non-empty resulting partition will be identical to the local index (sub)partition(s) of the partition that was split.

The same optimization holds for `SPLIT SUBPARTITION` operations.

Truncating Partitions

Use the `ALTER TABLE` ... `TRUNCATE PARTITION` statement to remove all rows from a table partition. Truncating a partition is similar to dropping a partition, except that the partition is emptied of its data, but not physically dropped.

You cannot truncate an index partition. However, if local indexes are defined for the table, the `ALTER TABLE` ... `TRUNCATE PARTITION` statement truncates the matching partition in each local index. Unless you specify `UPDATE INDEXES`, any global indexes are marked `UNUSABLE` and must be rebuilt. (You cannot use `UPDATE INDEXES` for index-organized tables. Use `UPDATE GLOBAL INDEXES` instead.)

Truncating a Table Partition

Use the `ALTER TABLE` ... `TRUNCATE PARTITION` statement to remove all rows from a table partition, with or without reclaiming space. Truncating a partition in an interval-partitioned table does not move the transition point. You can truncate partitions and subpartitions in a reference-partitioned table.

Truncating Table Partitions Containing Data and Global Indexes If the partition contains data and global indexes, use one of the following methods to truncate the table partition.

Method 1

Leave the global indexes in place during the `ALTER TABLE` ... `TRUNCATE PARTITION` statement. In this example, table `sales` has a global index `sales_area_ix`, which is rebuilt.

```
ALTER TABLE sales TRUNCATE PARTITION dec98;
ALTER INDEX sales_area_ix REBUILD;
```

This method is most appropriate for large tables where the partition being truncated contains a significant percentage of the total data in the table.

Method 2

Issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE` ... `TRUNCATE PARTITION` statement. The `DELETE` statement updates the global indexes, and also fires triggers and generates redo and undo logs.

For example, to truncate the first partition, issue the following statements:

```
DELETE FROM sales PARTITION (dec98);
ALTER TABLE sales TRUNCATE PARTITION dec98;
```

This method is most appropriate for small tables, or for large tables when the partition being truncated contains a small percentage of the total data in the table.

Method 3

Specify `UPDATE INDEXES` in the `ALTER TABLE` statement. This causes the global index to be truncated at the time the partition is truncated.

```
ALTER TABLE sales TRUNCATE PARTITION dec98
     UPDATE INDEXES;
```

Truncating a Partition Containing Data and Referential Integrity Constraints If a partition contains data and has referential integrity constraints, then you cannot truncate the partition. If no other data is referencing any data in the partition you want to remove, then choose either of the following methods to truncate the table partition.

Method 1

Disable the integrity constraints, issue the `ALTER TABLE` ... `TRUNCATE PARTITION` statement, then re-enable the integrity constraints. This method is most appropriate for large tables where the partition being truncated contains a significant percentage of the total data in the table. If there is still referencing data in other tables, then you have to remove that data in order to be able to re-enable the integrity constraints.

Method 2

Issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE` ... `TRUNCATE PARTITION` statement. The `DELETE` statement enforces referential integrity constraints, and also fires triggers and generates redo and undo logs. Data in referencing tables will be deleted if the foreign key constraints were created with the `ON DELETE CASCADE` option.

Note: You can substantially reduce the amount of logging by setting the `NOLOGGING` attribute (using `ALTER TABLE` ... `TRUNCATE PARTITION` ... `NOLOGGING`) for the partition before deleting all of its rows.

```
DELETE FROM sales partition (dec94);
ALTER TABLE sales TRUNCATE PARTITION dec94;
```

This method is most appropriate for small tables, or for large tables when the partition being truncated contains a small percentage of the total data in the table.

Truncating a Subpartition

You use the `ALTER TABLE` ... `TRUNCATE SUBPARTITION` statement to remove all rows from a subpartition of a composite partitioned table. Corresponding local index subpartitions are also truncated.

The following statement shows how to truncate data in a subpartition of a table. In this example, the space occupied by the deleted rows is made available for use by other schema objects in the tablespace.

```
ALTER TABLE diving
   TRUNCATE SUBPARTITION us_locations
      DROP STORAGE;
```

Dropping Partitioned Tables

Oracle Database processes a `DROP TABLE` statement for a partitioned table in the same way that it processes the statement for a non-partitioned table. One exception that was introduced in Oracle Database 10g Release 2 is when you use the `PURGE` keyword.

To avoid running into resource constraints, the `DROP TABLE`...`PURGE` statement for a partitioned table drops the table in multiple transactions, where each transaction drops a subset of the partitions or subpartitions and then commits. The table becomes completely dropped at the conclusion of the final transaction. This behavior comes with some changes to the `DROP TABLE` statement that you should be aware of.

First, if the `DROP TABLE`...`PURGE` statement fails, you can take corrective action, if any, and then reissue the statement. The statement resumes at the point where it failed.

Second, while the `DROP TABLE`...`PURGE` statement is in progress, the table is marked as unusable by setting the `STATUS` column to the value `UNUSABLE` in the following data dictionary views:

- `USER_TABLES`, `ALL_TABLES`, `DBA_TABLES`
- `USER_PART_TABLES`, `ALL_PART_TABLES`, `DBA_PART_TABLES`
- `USER_OBJECT_TABLES`, `ALL_OBJECT_TABLES`, `DBA_OBJECT_TABLES`

You can list all `UNUSABLE` partitioned tables by querying the `STATUS` column of these views.

Queries against other data dictionary views pertaining to partitioning, such as `DBA_TAB_PARTITIONS` and `DBA_TAB_SUBPARTITIONS`, exclude rows belonging to an `UNUSABLE` table. A complete list of these views is available in "Viewing Information About Partitioned Tables and Indexes" on page 3-69.

After a table is marked `UNUSABLE`, the only statement that can be issued against it is another `DROP TABLE`...`PURGE` statement, and only if the previous `DROP TABLE`...`PURGE` statement failed. Any other statement issued against an `UNUSABLE` table results in an error. The table remains in the `UNUSABLE` state until the drop operation is complete.

See Also:

- *Oracle Database SQL Language Reference* for the syntax of the `DROP TABLE` statement
- *Oracle Database Reference* for a description of the data dictionary views mentioned in this section.

Partitioned Tables and Indexes Example

This section presents an example of moving the time window in a historical table.

A **historical** table describes the business transactions of an enterprise over intervals of time. Historical tables can be **base** tables, which contain base information; for example, sales, checks, and orders. Historical tables can also be **rollup** tables, which contain summary information derived from the base information using operations such as `GROUP BY`, `AVERAGE`, or `COUNT`.

The time interval in a historical table is often a rolling window. DBAs periodically delete sets of rows that describe the oldest transactions, and in turn allocate space for sets of rows that describe the most recent transactions. For example, at the close of business on April 30, 1995, the DBA deletes the rows (and supporting index entries) that describe transactions from April 1994, and allocates space for the April 1995 transactions.

Now consider a specific example. You have a table, `order`, which contains 13 months of transactions: a year of historical data in addition to orders for the current month. There is one partition for each month. These monthly partitions are named `order_`*`yymm`*, as are the tablespaces in which they reside.

The `order` table contains two local indexes, `order_ix_onum`, which is a local, prefixed, unique index on the order number, and `order_ix_supp`, which is a local, non-prefixed index on the supplier number. The local index partitions are named with suffixes that match the underlying table. There is also a global unique index, `order_ix_cust`, for the customer name. `order_ix_cust` contains three partitions, one for each third of the alphabet. So on October 31, 1994, change the time window on `order` as follows:

1. Back up the data for the oldest time interval.

   ```
   ALTER TABLESPACE order_9310 BEGIN BACKUP;
   ...
   ALTER TABLESPACE order_9310 END BACKUP;
   ```

2. Drop the partition for the oldest time interval.

   ```
   ALTER TABLE order DROP PARTITION order_9310;
   ```

3. Add the partition to the most recent time interval.

   ```
   ALTER TABLE order ADD PARTITION order_9411;
   ```

4. Re-create the global index partitions.

   ```
   ALTER INDEX order_ix_cust REBUILD PARTITION order_ix_cust_AH;
   ALTER INDEX order_ix_cust REBUILD PARTITION order_ix_cust_IP;
   ALTER INDEX order_ix_cust REBUILD PARTITION order_ix_cust_QZ;
   ```

Ordinarily, the database acquires sufficient locks to ensure that no operation (DML, DDL, or utility) interferes with an individual DDL statement, such as `ALTER TABLE` ... `DROP PARTITION`. However, if the partition maintenance operation requires several steps, it is the database administrator's responsibility to ensure that applications (or other maintenance operations) do not interfere with the multistep operation in progress. Some methods for doing this are:

- Bring down all user-level applications during a well-defined batch window.
- Ensure that no one is able to access table `order` by revoking access privileges from a role that is used in all applications.

Viewing Information About Partitioned Tables and Indexes

The following views display information specific to partitioned tables and indexes:

View	Description

`DBA_PART_TABLES`
`ALL_PART_TABLES`
`USER_PART_TABLES`
	`DBA` view displays partitioning information for all partitioned tables in the database. `ALL` view displays partitioning information for all partitioned tables accessible to the user. `USER` view is restricted to partitioning information for partitioned tables owned by the user.

`DBA_TAB_PARTITIONS`
`ALL_TAB_PARTITIONS`
`USER_TAB_PARTITIONS`
	Display partition-level partitioning information, partition storage parameters, and partition statistics generated by the `DBMS_STATS` package or the `ANALYZE` statement.

View	Description
DBA_TAB_SUBPARTITIONS ALL_TAB_SUBPARTITIONS USER_TAB_SUBPARTITIONS	Display subpartition-level partitioning information, subpartition storage parameters, and subpartition statistics generated by the DBMS_ STATS package or the ANALYZE statement.
DBA_PART_KEY_COLUMNS ALL_PART_KEY_COLUMNS USER_PART_KEY_COLUMNS	Display the partitioning key columns for partitioned tables.
DBA_SUBPART_KEY_COLUMNS ALL_SUBPART_KEY_COLUMNS USER_SUBPART_KEY_COLUMNS	Display the subpartitioning key columns for composite-partitioned tables (and local indexes on composite-partitioned tables).
DBA_PART_COL_STATISTICS ALL_PART_COL_STATISTICS USER_PART_COL_STATISTICS	Display column statistics and histogram information for the partitions of tables.
DBA_SUBPART_COL_STATISTICS ALL_SUBPART_COL_STATISTICS USER_SUBPART_COL_STATISTICS	Display column statistics and histogram information for subpartitions of tables.
DBA_PART_HISTOGRAMS ALL_PART_HISTOGRAMS USER_PART_HISTOGRAMS	Display the histogram data (end-points for each histogram) for histograms on table partitions.
DBA_SUBPART_HISTOGRAMS ALL_SUBPART_HISTOGRAMS USER_SUBPART_HISTOGRAMS	Display the histogram data (end-points for each histogram) for histograms on table subpartitions.
DBA_PART_INDEXES ALL_PART_INDEXES USER_PART_INDEXES	Display partitioning information for partitioned indexes.
DBA_IND_PARTITIONS ALL_IND_PARTITIONS USER_IND_PARTITIONS	Display the following for index partitions: partition-level partitioning information, storage parameters for the partition, statistics collected by the DBMS_STATS package or the ANALYZE statement.
DBA_IND_SUBPARTITIONS ALL_IND_SUBPARTITIONS USER_IND_SUBPARTITIONS	Display the following information for index subpartitions: partition-level partitioning information, storage parameters for the partition, statistics collected by the DBMS_STATS package or the ANALYZE statement.
DBA_SUBPARTITION_TEMPLATES ALL_SUBPARTITION_TEMPLATES USER_SUBPARTITION_TEMPLATES	Display information about existing subpartition templates.

See Also:

- *Oracle Database Reference* for complete descriptions of these views
- *Oracle Database Performance Tuning Guide* and *Oracle Database Performance Tuning Guide* for information about histograms and generating statistics for tables
- *Oracle Database Administrator's Guide* for more information about analyzing tables, indexes, and clusters

4

Partitioning for Availability, Manageability, and Performance

This chapter provides high-level insight into how partitioning enables availability, manageability, and performance. This chapter presents guidelines on when to use a given partitioning strategy. The main focus of this chapter is the use of table partitioning, though most of the recommendations and considerations apply to index partitioning as well.

This chapter contains the following topics:

Partition Pruning

Partition pruning is an essential performance feature for data warehouses. In partition pruning, the optimizer analyzes `FROM` and `WHERE` clauses in SQL statements to eliminate unneeded partitions when building the partition access list. This enables Oracle Database to perform operations only on those partitions that are relevant to the SQL statement.

Partition pruning dramatically reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance and optimizing resource utilization. If you partition the index and table on different columns (with a global partitioned index), then partition pruning also eliminates index partitions even when the partitions of the underlying table cannot be eliminated.

Depending upon the actual SQL statement, Oracle Database may use static or dynamic pruning. Static pruning occurs at compile-time, with the information about the partitions accessed beforehand while dynamic pruning occurs at run-time, meaning that the exact partitions to be accessed by a statement are not known beforehand. A sample scenario for static pruning would be a SQL statement containing a `WHERE` condition with a constant literal on the partition key column. An example of dynamic pruning is the use of operators or functions in the `WHERE` condition.

Partition pruning affects the statistics of the objects where pruning will occur and will therefore also affect the execution plan of a statement.

Information that can be Used for Partition Pruning

Oracle Database prunes partitions when you use range, `LIKE`, equality, and `IN`-list predicates on the range or list partitioning columns, and when you use equality and `IN`-list predicates on the hash partitioning columns.

On composite partitioned objects, Oracle can prune at both levels using the relevant predicates. Examine the table `sales_range_hash`, which is partitioned by range on the column `s_saledate` and subpartitioned by hash on the column `s_productid`, and consider the following example:

```
CREATE TABLE sales_range_hash(
  s_productid  NUMBER,
  s_saledate   DATE,
  s_custid     NUMBER,
  s_totalprice NUMBER)
PARTITION BY RANGE (s_saledate)
SUBPARTITION BY HASH (s_productid) SUBPARTITIONS 8
 (PARTITION sal99q1 VALUES LESS THAN
   (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
  PARTITION sal99q2 VALUES LESS THAN
   (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
  PARTITION sal99q3 VALUES LESS THAN
   (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
  PARTITION sal99q4 VALUES LESS THAN
   (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));

SELECT * FROM sales_range_hash
WHERE s_saledate BETWEEN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY'))
  AND (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')) AND s_productid = 1200;
```

Oracle uses the predicate on the partitioning columns to perform partition pruning as follows:

- When using range partitioning, Oracle accesses only partitions `sal99q2` and `sal99q3`, representing the partitions for the third and fourth quarters of 1999.
- When using hash subpartitioning, Oracle accesses only the one subpartition in each partition that stores the rows with `s_productid=1200`. The mapping between the subpartition and the predicate is calculated based on Oracle's internal hash distribution function.

A reference-partitioned table can take advantage of partition pruning through the join with the referenced table. Virtual column-based partitioned tables benefit from partition pruning for statements that use the virtual column-defining expression in the SQL statement.

How to Identify Whether Partition Pruning has been Used

Whether Oracle uses partition pruning or not is reflected in the execution plan of a statement, either in the plan table for the `EXPLAIN PLAN` statement or in the shared SQL area.

The partition pruning information is reflected in the plan columns `PSTART` (`PARTITION_START`) and `PSTOP` (`PARTITION_STOP`). In the case of serial statements, the pruning information is also reflected in the `OPERATION` and `OPTIONS` columns.

See Also: *Oracle Database Performance Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

Static Partition Pruning

For a number of cases, Oracle determines the partitions to be accessed at compile time. Static partition pruning will occur if you use static predicates, except for the following cases:

- Partition pruning occurs using the result of a subquery.
- The optimizer rewrites the query with a star transformation and pruning occurs after the star transformation.
- The most efficient execution plan is a nested loop.

These three cases result in the use of dynamic pruning.

If at parse time Oracle can identify which contiguous set of partitions will be accessed, then the `PSTART` and `PSTOP` columns in the execution plan will show the begin and the end values of the partitions being accessed. Any other cases of partition pruning, including dynamic pruning, will show the KEY value in `PSTART` and `PSTOP`, optionally with an additional attribute.

The following is an example:

```
SQL> explain plan for select * from sales where time_id = to_date('01-jan-2001', 'dd-mon-yyyy');
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------
Plan hash value: 3971874201
----------------------------------------------------------------------------------------------
| Id | Operation               | Name  | Rows | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT        |       | 673  | 19517 | 27      (8)| 00:00:01 |       |       |
|  1 |  PARTITION RANGE SINGLE|       | 673  | 19517 | 27      (8)| 00:00:01 | 17    | 17    |
|* 2 |   TABLE ACCESS FULL     | SALES | 673  | 19517 | 27      (8)| 00:00:01 | 17    | 17    |
----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("TIME_ID"=TO_DATE('2001-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

This plan shows that Oracle accesses partition number 17, as shown in the `PSTART` and `PSTOP` columns. The `OPERATION` column shows `PARTITION RANGE SINGLE`, indicating that only a single partition is being accessed. If `OPERATION` shows `PARTITION RANGE ALL`, then all partitions are being accessed and effectively no pruning takes place. `PSTART` then shows the very first partition of the table and `PSTOP` shows the very last partition.

An execution plan with a full table scan on an interval-partitioned table shows 1 for `PSTART`, and 1048575 for `PSTOP`, regardless of how many interval partitions were created.

Dynamic Partition Pruning

Dynamic pruning occurs if pruning is possible and static pruning is not possible. The following examples show a number of dynamic pruning cases.

Dynamic Pruning with Bind Variables

Statements that use bindvariables against partition columns result in dynamic pruning. For example:

```
SQL> explain plan for select * from sales s where time_id in ( :a, :b, :c, :d);
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------------------------
Plan hash value: 513834092
---------------------------------------------------------------------------------------------------
| Id | Operation                         |     Name |Rows|Bytes|Cost (%CPU)|  Time    | Pstart| Pstop|
---------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                  |          |2517|72993|    292 (0)|00:00:04|        |       |
|  1 |  INLIST ITERATOR                  |          |    |     |           |        |        |       |
|  2 |   PARTITION RANGE ITERATOR        |          |2517|72993|    292 (0)|00:00:04|KEY(I)  |KEY(I)|
|  3 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES  |2517|72993|    292 (0)|00:00:04|KEY(I)  |KEY(I)|
|  4 |     BITMAP CONVERSION TO ROWIDS   |          |    |     |           |        |        |       |
|* 5 |      BITMAP INDEX SINGLE VALUE    |SALES_TIME_BIX| |   |           |        |KEY(I)  |KEY(I)|
---------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
5 - access("TIME_ID"=:A OR "TIME_ID"=:B OR "TIME_ID"=:C OR "TIME_ID"=:D)
```

For parallel plans, only the partition start and stop columns contain the partition pruning information; the operation column will contain information for the parallel operation, as shown in the following example:

```
SQL> explain plan for select * from sales where time_id in (:a, :b, :c, :d);
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------
Plan hash value: 4058105390
-------------------------------------------------------------------------------------------------
| Id| Operation          | Name  |Rows|Bytes|Cost(%CP|  Time   |Pstart| Pstop|  TQ |INOUT| PQ Dis|
-------------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT   |       |2517|72993|  75(36)|00:00:01|      |      |     |     |       |
|  1|  PX COORDINATOR    |       |    |     |        |        |      |      |     |     |       |
|  2|  PX SEND QC(RANDOM)|:TQ10000|2517|72993| 75(36)|00:00:01|      |      |Q1,00| P->S|QC(RAND|
|  3|   PX BLOCK ITERATOR|       |2517|72993|  75(36)|00:00:01|KEY(I)|KEY(I)|Q1,00| PCWC|       |
|* 4|   TABLE ACCESS FULL| SALES |2517|72993|  75(36)|00:00:01|KEY(I)|KEY(I)|Q1,00| PCWP|       |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  4 - filter("TIME_ID"=:A OR "TIME_ID"=:B OR "TIME_ID"=:C OR "TIME_ID"=:D)
```

See Also: *Oracle Database Performance Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

Dynamic Pruning with Subqueries

Statements that explicitely use subqueries against partition columns result in dynamic pruning. For example:

```
SQL> explain plan for select sum(amount_sold) from sales where time_id in
     (select time_id from times where fiscal_year = 2000);
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------
```

```
Plan hash value: 3827742054

----------------------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |     1 |    25 |   523   (5)| 00:00:07 |       |       |
|   1 |  SORT AGGREGATE           |       |     1 |    25 |            |          |       |       |
|*  2 |   HASH JOIN               |       |  191K|  4676K|   523   (5)| 00:00:07 |       |       |
|*  3 |    TABLE ACCESS FULL      | TIMES |   304 |  3648 |    18   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE SUBQUERY|       |  918K|    11M|   498   (4)| 00:00:06 |KEY(SQ)|KEY(SQ)|
|   5 |     TABLE ACCESS FULL     | SALES |  918K|    11M|   498   (4)| 00:00:06 |KEY(SQ)|KEY(SQ)|
----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("TIME_ID"="TIME_ID")
   3 - filter("FISCAL_YEAR"=2000)
```

See Also: *Oracle Database Performance Tuning Guide* for more information about EXPLAIN PLAN and how to interpret it

Dynamic Pruning with Star Transformation

Statements that get transformed by the database using the star transformation result in dynamic pruning. For example:

```
SQL> explain plan for select p.prod_name, t.time_id, sum(s.amount_sold)
     from sales s, times t, products p
     where s.time_id = t.time_id and s.prod_id = p.prod_id and t.fiscal_year = 2000
     and t.fiscal_week_number = 3 and p.prod_category = 'Hardware'
     group by t.time_id, p.prod_name;
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
Plan hash value: 4020965003

------------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name                  | Rows  | Bytes | Pstart| Pstop |
------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                       |     1 |    79 |       |       |
|   1 |  HASH GROUP BY                       |                       |     1 |    79 |       |       |
|*  2 |   HASH JOIN                          |                       |     1 |    79 |       |       |
|*  3 |    HASH JOIN                         |                       |     2 |    64 |       |       |
|*  4 |     TABLE ACCESS FULL                | TIMES                 |     6 |    90 |       |       |
|   5 |     PARTITION RANGE SUBQUERY         |                       |   587 |  9979 |KEY(SQ)|KEY(SQ)|
|   6 |      TABLE ACCESS BY LOCAL INDEX ROWID| SALES                 |   587 |  9979 |KEY(SQ)|KEY(SQ)|
|   7 |       BITMAP CONVERSION TO ROWIDS    |                       |       |       |       |       |
|   8 |        BITMAP AND                    |                       |       |       |       |       |
|   9 |         BITMAP MERGE                 |                       |       |       |       |       |
|  10 |          BITMAP KEY ITERATION        |                       |       |       |       |       |
|  11 |           BUFFER SORT                |                       |       |       |       |       |
|* 12 |            TABLE ACCESS FULL         | TIMES                 |     6 |    90 |       |       |
|* 13 |           BITMAP INDEX RANGE SCAN    | SALES_TIME_BIX        |       |       |KEY(SQ)|KEY(SQ)|
|  14 |         BITMAP MERGE                 |                       |       |       |       |       |
|  15 |          BITMAP KEY ITERATION        |                       |       |       |       |       |
|  16 |           BUFFER SORT                |                       |       |       |       |       |
|  17 |            TABLE ACCESS BY INDEX ROWID| PRODUCTS             |    14 |   658 |       |       |
|* 18 |             INDEX RANGE SCAN         | PRODUCTS_PROD_CAT_IX  |    14 |       |       |       |
|* 19 |           BITMAP INDEX RANGE SCAN    | SALES_PROD_BIX        |       |       |KEY(SQ)|KEY(SQ)|
|  20 |    TABLE ACCESS BY INDEX ROWID       | PRODUCTS              |    14 |   658 |       |       |
|* 21 |     INDEX RANGE SCAN                 | PRODUCTS_PROD_CAT_IX  |    14 |       |       |       |
------------------------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("S"."PROD_ID"="P"."PROD_ID")
   3 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter("T"."FISCAL_WEEK_NUMBER"=3 AND "T"."FISCAL_YEAR"=2000)
  12 - filter("T"."FISCAL_WEEK_NUMBER"=3 AND "T"."FISCAL_YEAR"=2000)
  13 - access("S"."TIME_ID"="T"."TIME_ID")
  18 - access("P"."PROD_CATEGORY"='Hardware')
  19 - access("S"."PROD_ID"="P"."PROD_ID")
  21 - access("P"."PROD_CATEGORY"='Hardware')

Note
-----
   - star transformation used for this statement
```

Note: The `Cost (%CPU)` and `Time` columns were removed from the plan table output in this example.

See Also: *Oracle Database Performance Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

Dynamic Pruning with Nested Loop Joins

Statements that are most efficiently executed using a nested loop join use dynamic pruning. For example:

```
SQL> explain plan for select t.time_id, sum(s.amount_sold)
     from sales s, times t
     where s.time_id = t.time_id and t.fiscal_year = 2000 and t.fiscal_week_number = 3
     group by t.time_id;
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
----------------------------------------------------------------------------------------------------
Plan hash value: 50737729

----------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     6 |   168 |   126   (4)| 00:00:02 |       |       |
|   1 |  HASH GROUP BY             |       |     6 |   168 |   126   (4)| 00:00:02 |       |       |
|   2 |   NESTED LOOPS             |       |  3683 |   100K|   125   (4)| 00:00:02 |       |       |
|*  3 |    TABLE ACCESS FULL       | TIMES |     6 |    90 |    18   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ITERATOR|       |   629 |  8177 |    18   (6)| 00:00:01 |   KEY |   KEY |
|*  5 |     TABLE ACCESS FULL      | SALES |   629 |  8177 |    18   (6)| 00:00:01 |   KEY |   KEY |
----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("T"."FISCAL_WEEK_NUMBER"=3 AND "T"."FISCAL_YEAR"=2000)
   5 - filter("S"."TIME_ID"="T"."TIME_ID")
```

See Also: *Oracle Database Performance Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

Partition Pruning Tips

When using partition pruning, you should consider the following:

- Datatype Conversions
- Function Calls

Datatype Conversions

In order to get the maximum performance benefit from partition pruning, you should avoid constructs that require the database to convert the datatype you specify. Datatype conversions typically result in dynamic pruning when static pruning would have otherwise been possible. SQL statements that benefit from static pruning perform better than statements that benefit from dynamic pruning.

A common case of datatype conversions occurs when using the Oracle `DATE` datatype. An Oracle `DATE` datatype is not a character string but is only represented as such when querying the database; the format of the representation is defined by the NLS setting of the instance or the session. Consequently, the same reverse conversion has to happen when inserting data into a `DATE` field or specifying a predicate on such a field.

A conversion can either happen implicitly or explicitly by specifying a `TO_DATE` conversion. Only a properly applied `TO_DATE` function guarantees that the database is capable of uniquely determining the date value and using it potentially for static pruning, which is especially beneficial for single partition access.

Consider the following example that runs against the sample `SH` schema in an Oracle Database. You would like to know the total revenue number for the year 2000. There are multiple ways you can retrieve the answer to the query, but not every method is equally efficient.

```
explain plan for SELECT SUM(amount_sold) total_revenue
FROM sales,
WHERE time_id between '01-JAN-00' and '31-DEC-00';
```

The plan should now be similar to the following:

```
----------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name   | Rows   | Bytes  | Cost (%CPU)| Time      | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |        |      1 |     13 |   525   (8)| 00:00:07 |       |       |
|   1 |  SORT AGGREGATE             |        |      1 |     13 |            |           |       |       |
|*  2 |   FILTER                    |        |        |        |            |           |       |       |
|   3 |    PARTITION RANGE ITERATOR|        |   230K|   2932K|   525   (8)| 00:00:07 |   KEY |   KEY |
|*  4 |     TABLE ACCESS FULL       | SALES  |   230K|   2932K|   525   (8)| 00:00:07 |   KEY |   KEY |
----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('01-JAN-00')<=TO_DATE('31-DEC-00'))
   4 - filter("TIME_ID">='01-JAN-00' AND "TIME_ID"<='31-DEC-00')
```

In this case, the keyword `KEY` for both `PSTART` and `PSTOP` means that dynamic partition pruning occurs at run-time. Consider the following case.

```
explain plan for select sum(amount_sold)
from sales
where time_id between '01-JAN-2000' and '31-DEC-2000' ;
```

The execution plan now shows the following:

```
---------------------------------------------------------------------------------------------------
```

```
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Pstart| Pstop |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |    13 |   127   (4)|       |       |
|   1 |  SORT AGGREGATE            |       |     1 |    13 |            |       |       |
|   2 |   PARTITION RANGE ITERATOR|       |  230K|  2932K|   127   (4)|    13 |    16 |
|*  3 |    TABLE ACCESS FULL       | SALES |  230K|  2932K|   127   (4)|    13 |    16 |
----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("TIME_ID"<=TO_DATE(' 2000-12-31 00:00:00', "syyyy-mm-dd hh24:mi:ss'))
```

Note: The `Time` column was removed from the execution plan.

The execution plan shows static partition pruning. The query accesses a contiguous list of partitions 13 to 16. In this particular case the way the date format was specified matches the NLS date format setting. Though this example shows the most efficient execution plan, you cannot rely on the NLS date format setting to define a certain format.

```
alter session set nls_date_format='fmdd Month yyyy';

explain plan for select sum(amount_sold)
from sales
where time_id between '01-JAN-2000' and '31-DEC-2000' ;
```

The execution plan now shows the following:

```
----------------------------------------------------------------------------------------
| Id  | Operation                   | Name  | Rows  | Bytes | Cost (%CPU)| Pstart| Pstop |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |       |     1 |    13 |   525   (8)|       |       |
|   1 |  SORT AGGREGATE             |       |     1 |    13 |            |       |       |
|*  2 |   FILTER                    |       |       |       |            |       |       |
|   3 |    PARTITION RANGE ITERATOR|       |  230K|  2932K|   525   (8)|   KEY |   KEY |
|*  4 |     TABLE ACCESS FULL       | SALES |  230K|  2932K|   525   (8)|   KEY |   KEY |
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('01-JAN-2000')<=TO_DATE('31-DEC-2000'))
   4 - filter("TIME_ID">='01-JAN-2000' AND "TIME_ID"<='31-DEC-2000')
```

Note: The `Time` column was removed from the execution plan.

This plan, which uses dynamic pruning, again is less efficient than the static pruning execution plan. In order to guarantee a static partition pruning plan, you should explicitly convert datatypes to match the partition column datatype. For example:

```
explain plan for select sum(amount_sold)
from sales
where time_id between to_date('01-JAN-2000','dd-MON-yyyy')
  and to_date('31-DEC-2000','dd-MON-yyyy') ;
```

```
----------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Pstart| Pstop |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |    13 |   127   (4)|       |       |
|   1 |  SORT AGGREGATE            |       |     1 |    13 |            |       |       |
|   2 |   PARTITION RANGE ITERATOR|       |   230K|  2932K|   127   (4)|    13 |    16 |
|*  3 |    TABLE ACCESS FULL       | SALES |   230K|  2932K|   127   (4)|    13 |    16 |
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("TIME_ID"<=TO_DATE(' 2000-12-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Note: The `Time` column was removed from the execution plan.

See Also:

- *Oracle Database SQL Language Reference* for details about the `DATE` datatype
- *Oracle Database Globalization Support Guide* for details about NLS settings and globalization issues

Function Calls

There are several cases when the optimizer cannot perform any pruning. One of the most common reasons is when an operator is used on top of a partitioning column. This could be an explicit operator (for example, a function) or even an implicit operator introduced by Oracle as part of the necessary data type conversion for executing the statement. For example, consider the following query:

```
EXPLAIN PLAN FOR
SELECT SUM(quantity_sold)
FROM sales
WHERE time_id = TO_TIMESTAMP('1-jan-2000', 'dd-mon-yyyy');
```

Because `time_id` is of type `DATE` and Oracle needs to promote it to the `TIMESTAMP` type to get the same datatype, this predicate is internally rewritten as:

```
TO_TIMESTAMP(time_id) = TO_TIMESTAMP('1-jan-2000', 'dd-mon-yyyy')
```

The explain plan for this statement is as follows:

```
--------------------------------------------------------------------------------------------
|Id | Operation              | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
--------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT       |       |     1 |    11 |      6  (17)| 00:00:01 |       |       |
| 1 |  SORT AGGREGATE        |       |     1 |    11 |            |          |       |       |
| 2 |   PARTITION RANGE ALL|       |    10 |   110 |      6  (17)| 00:00:01 |     1 |    16 |
|*3 |    TABLE ACCESS FULL | SALES |    10 |   110 |      6  (17)| 00:00:01 |     1 |    16 |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
3 - filter(INTERNAL_FUNCTION("TIME_ID")=TO_TIMESTAMP('1-jan-2000',:B1))

15 rows selected
```

The SELECT statement accesses all partitions even though pruning down to a single partition could have taken place. Consider the example to find the total sales revenue number for 2000. Another way to construct the query would be:

```
explain plan for select sum(amount_sold)
from sales
where to_char(time_id,'yyyy') = '2000';
```

This query applies a function call to the partition key column, which generally disables partition pruning. The execution plan shows a full table scan with no partition pruning:

```
----------------------------------------------------------------------------------------------
| Id  | Operation              | Name  | Rows  | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |       |     1 |    13 |   527   (9)| 00:00:07 |       |       |
|   1 |  SORT AGGREGATE        |       |     1 |    13 |            |          |       |       |
|   2 |   PARTITION RANGE ALL|       |  9188 |   116K|   527   (9)| 00:00:07 |     1 |    28 |
|*  3 |    TABLE ACCESS FULL | SALES |  9188 |   116K|   527   (9)| 00:00:07 |     1 |    28 |
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(TO_CHAR(INTERNAL_FUNCTION("TIME_ID"),'yyyy')='2000')
```

Avoid using implicit or explicit functions on the partition columns. If your queries commonly use function calls, then consider using a virtual column and virtual column partitioning to benefit from partition pruning in these cases.

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 the use of both CPU and memory resources. In Oracle Real Application Clusters (RAC) environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which is the key to achieving good scalability for massive join operations.

Partition-wise joins can be full or partial. Oracle decides which type of join to use.

Full Partition-Wise Joins

A full partition-wise join divides a large join into smaller joins between a pair of partitions from the two joined tables. To use this feature, you must equipartition both tables on their join keys, or use reference partitioning. For example, consider a large join between a sales table and a customer table on the column customerid. The query "find the records of all customers who bought more than 100 articles in Quarter 3 of 1999" is a typical example of a SQL statement performing such a join. The following is an example of this:

```
SELECT c.cust_last_name, COUNT(*)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND
s.time_id BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY') AND
     (TO_DATE('01-OCT-1999', 'DD-MON-YYYY'))
GROUP BY c.cust_last_name HAVING COUNT(*) > 100;
```

Such a large join is typical in data warehousing environments. In this case, the entire customer table is joined with one quarter of the sales data. In large data warehouse applications, this might mean joining millions of rows. The join method to use in that case is obviously a hash join. You can reduce the processing time for this hash join even more if both tables are equipartitioned on the `cust_id` column. This enables a full partition-wise join.

When you execute a full partition-wise join in parallel, the granule of parallelism is a partition. As a result, the degree of parallelism is limited to the number of partitions. For example, you require at least 16 partitions to set the degree of parallelism of the query to 16.

You can use various partitioning methods to equipartition both tables. These methods are described at a high level in the following subsections.

Full Partition-Wise Joins: Single-Level - Single-Level

This is the simplest method: two tables are both partitioned by the join column. In the example, the `customers` and `sales` tables are both partitioned on the `cust_id` columns. This partitioning method enables full partition-wise joins when the tables are joined on `cust_id`, both representing the same customer identification number. This scenario is available for range-range, list-list, and hash-hash partitioning. Interval-range and interval-interval full partition-wise joins are also supported and can be compared to range-range.

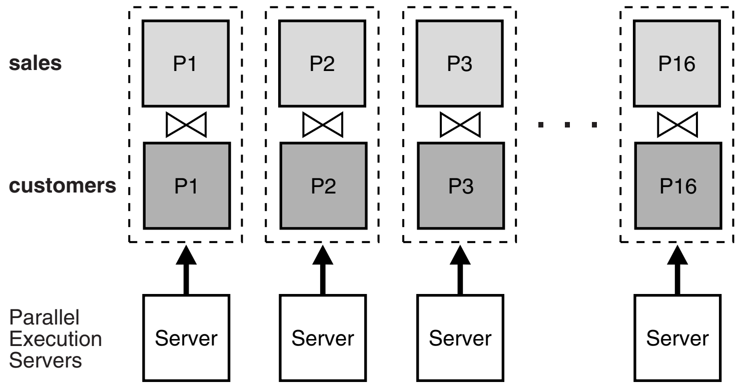
In serial, this join is performed between pairs of matching hash partitions, one at a time. When one partition pair has been joined, the join of another partition pair begins. The join completes when all partition pairs have been processed. To ensure a good workload distribution, you should either have many more partitions than the requested degree of parallelism or use equi-size partitions with as many partitions as the requested degree of parallelism. Using hash partitioning on a unique or almost-unique column, with the number of partitions equal to a power of 2, is a good way to create equi-sized partitions.

Note:

- A pair of matching hash partitions is defined as one partition with the same partition number from each table. For example, with full partition-wise joins based on hash partitioning, the database joins partition 0 of `sales` with partition 0 of `customers`, partition 1 of `sales` with partition 1 of `customers`, and so on.
- Reference partitioning is an easy way to co-partition two tables so that the optimizer can always consider a full partition-wise join if the tables are joined in a statement.

Parallel execution of a full partition-wise join is a straightforward parallelization of the serial execution. Instead of joining one partition pair at a time, partition pairs are joined in parallel by the query servers. Figure 4–1 illustrates the parallel execution of a full partition-wise join.

Figure 4–1 Parallel Execution of a Full Partition-wise Join



The following example shows the execution plan for `sales` and `customers` co-partitioned by hash with the same number of partitions. The plan shows a full partition-wise join.

```
explain plan for SELECT c.cust_last_name, COUNT(*)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND
s.time_id BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY') AND
     (TO_DATE('01-OCT-1999', 'DD-MON-YYYY'))
GROUP BY c.cust_last_name HAVING COUNT(*) > 100;

---------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name       | Rows  | Bytes  | Pstart| Pstop |    TQ   |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |    46 |  1196  |       |       |         |      |            |
|   1 |  PX COORDINATOR               |            |       |        |       |       |         |      |            |
|   2 |   PX SEND QC (RANDOM)         | :TQ10001   |    46 |  1196  |       |       |  Q1,01  | P->S | QC (RAND)  |
|*  3 |    FILTER                     |            |       |        |       |       |  Q1,01  | PCWC |            |
|   4 |     HASH GROUP BY             |            |    46 |  1196  |       |       |  Q1,01  | PCWP |            |
|   5 |      PX RECEIVE               |            |    46 |  1196  |       |       |  Q1,01  | PCWP |            |
|   6 |       PX SEND HASH            | :TQ10000   |    46 |  1196  |       |       |  Q1,00  | P->P | HASH       |
|   7 |        HASH GROUP BY          |            |    46 |  1196  |       |       |  Q1,00  | PCWP |            |
|   8 |         PX PARTITION HASH ALL|            | 59158 |  1502K|     1 |    16 |  Q1,00  | PCWC |            |
|*  9 |          HASH JOIN            |            | 59158 |  1502K|       |       |  Q1,00  | PCWP |            |
|  10 |           TABLE ACCESS FULL   | CUSTOMERS  | 55500 |   704K|     1 |    16 |  Q1,00  | PCWP |            |
|* 11 |           TABLE ACCESS FULL   | SALES      | 59158 |   751K|     1 |    16 |  Q1,00  | PCWP |            |
---------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   9 - access("S"."CUST_ID"="C"."CUST_ID")
  11 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Note: The `Cost (%CPU)` and `Time` columns were removed from the plan table output in this example.

In Oracle Real Application Clusters environments running on MPP platforms, placing partitions on nodes is critical to achieving good scalability. To avoid remote I/O, both matching partitions should have affinity to the same node. Partition pairs should be spread over all nodes to avoid bottlenecks and to use all CPU resources available on the system.

Nodes can host multiple pairs when there are more pairs than nodes. For example, with an 8-node system and 16 partition pairs, each node receives two pairs.

> **See Also:** *Oracle Real Application Clusters Administration and Deployment Guide* for more information on data affinity

Full Partition-Wise Joins: Composite - Single-Level

This method is a variation of the single-level - single-level method. In this scenario, one table (typically the larger table) is composite partitioned on two dimensions, using the join columns as the subpartition key. In the example, the `sales` table is a typical example of a table storing historical data. Using range partitioning is a logical initial partitioning method for a table storing historical information.

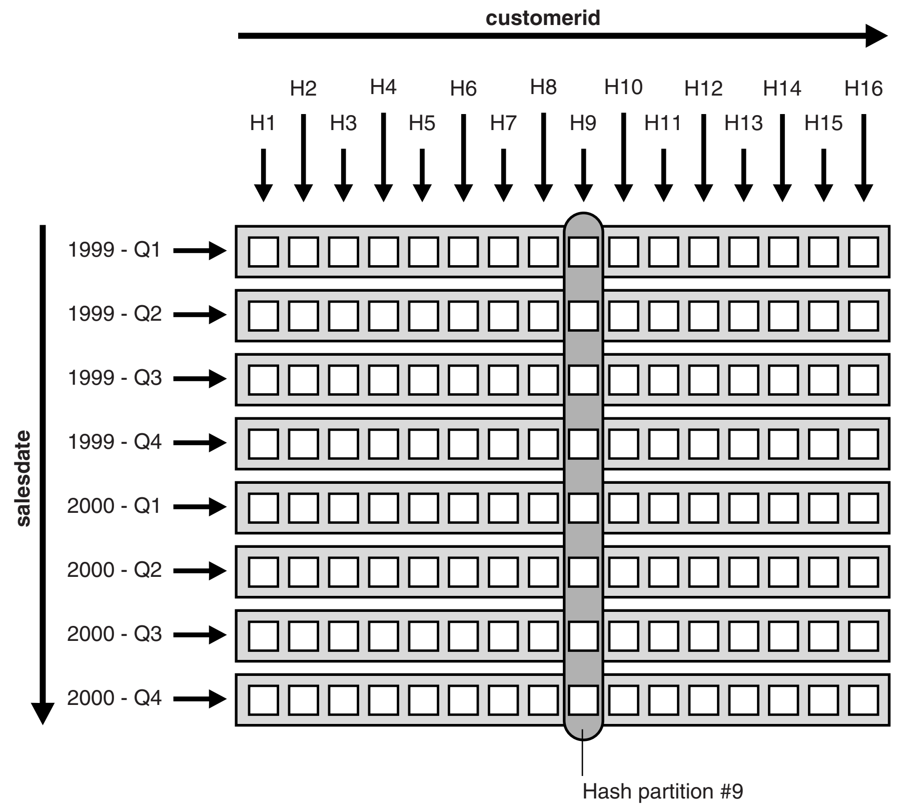
For example, assume you want to partition the `sales` table into eight partitions by range on the column `time_id`. Also assume you have two years and that each partition represents a quarter. Instead of using range partitioning, you can use composite partitioning to enable a full partition-wise join while preserving the partitioning on `time_id`. For example, partition the `sales` table by range on `time_id` and then subpartition each partition by hash on `cust_id` using 16 subpartitions for each partition, for a total of 128 subpartitions. The `customers` table can use hash partitioning with 16 partitions.

When you use the method just described, a full partition-wise join works similarly to the one created by a single-level - single-level hash-hash method. The join is still divided into 16 smaller joins between hash partition pairs from both tables. The difference is that now each hash partition in the `sales` table is composed of a set of 8 subpartitions, one from each range partition.

Figure 4–2 illustrates how the hash partitions are formed in the `sales` table. Each cell represents a subpartition. Each row corresponds to one range partition, for a total of 8 range partitions. Each range partition has 16 subpartitions. Each column corresponds to one hash partition for a total of 16 hash partitions; each hash partition has 8 subpartitions. Note that hash partitions can be defined only if all partitions have the same number of subpartitions, in this case, 16.

Hash partitions are implicit in a composite table. However, Oracle does not record them in the data dictionary, and you cannot manipulate them with DDL commands as you can range or list partitions.

Figure 4–2 Range and Hash Partitions of a Composite Table



The following example shows the execution plan for the full partition-wise join with the `sales` table range partitioned by `time_id`, and subpartitioned by hash on `cust_id`.

```
----------------------------------------------------------------------------------------------
| Id  | Operation                             | Name        | Pstart| Pstop |IN-OUT| PQ Distrib |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |             |       |       |      |            |
|   1 |  PX COORDINATOR                       |             |       |       |      |            |
|   2 |   PX SEND QC (RANDOM)                 | :TQ10001    |       |       | P->S | QC (RAND)  |
|*  3 |    FILTER                             |             |       |       | PCWC |            |
|   4 |     HASH GROUP BY                     |             |       |       | PCWP |            |
|   5 |      PX RECEIVE                       |             |       |       | PCWP |            |
|   6 |       PX SEND HASH                    | :TQ10000    |       |       | P->P | HASH       |
|   7 |        HASH GROUP BY                  |             |       |       | PCWP |            |
|   8 |         PX PARTITION HASH ALL         |             |     1 |    16 | PCWC |            |
|*  9 |          HASH JOIN                    |             |       |       | PCWP |            |
|  10 |           TABLE ACCESS FULL           | CUSTOMERS   |     1 |    16 | PCWP |            |
|  11 |           PX PARTITION RANGE ITERATOR|             |     8 |     9 | PCWC |            |
|* 12 |            TABLE ACCESS FULL          | SALES       |   113 |   144 | PCWP |            |
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   9 - access("S"."CUST_ID"="C"."CUST_ID")
  12 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Note: The `Rows`, `Cost (%CPU)`, `Time`, and `TQ` columns were removed from the plan table output in this example.

Composite - single-level partitioning is effective because it lets you combine pruning on one dimension with a full partition-wise join on another dimension. In the previous example query, pruning is achieved by scanning only the subpartitions corresponding to Q3 of 1999, in other words, row number 3 in Figure 4–2. Oracle then joins these subpartitions with the customer table, using a full partition-wise join.

All characteristics of the single-level - single-level partition-wise join apply to the composite - single-level partition-wise join. In particular, for this example, these two points are common to both methods:

- The degree of parallelism for this full partition-wise join cannot exceed 16. Even though the `sales` table has 128 subpartitions, it has only 16 hash partitions.
- The rules for data placement on MPP systems apply here. The only difference is that a subpartition is now a collection of subpartitions. You must ensure that all these subpartitions are placed on the same node as the matching hash partition from the other table. For example, in Figure 4–2, store hash partition 9 of the `sales` table shown by the eight circled subpartitions, on the same node as hash partition 9 of the `customers` table.

Full Partition-Wise Joins: Composite - Composite

If needed, you can also partition the `customer` table by the composite method. For example, you partition it by range on a postal code column to enable pruning based on postal codes. You then subpartition it by hash on `cust_id` using the same number of partitions (16) to enable a partition-wise join on the hash dimension.

You can get full partition-wise joins on all combinations of partition and subpartition partitions: partition - partition, partition - subpartition, subpartition - partition, and subpartition - subpartition.

Partial Partition-Wise Joins

Oracle Database can perform partial partition-wise joins only in parallel. Unlike full partition-wise joins, partial partition-wise joins require you to partition only one table on the join key, not both tables. The partitioned table is referred to as the reference table. The other table may or may not be partitioned. Partial partition-wise joins are more common than full partition-wise joins.

To execute a partial partition-wise join, the database dynamically repartitions the other table based on the partitioning of the reference table. Once the other table is repartitioned, the execution is similar to a full partition-wise join.

The performance advantage that partial partition-wise joins have over joins in non-partitioned tables is that the reference table is not moved during the join operation. Parallel joins between non-partitioned tables require both input tables to be redistributed on the join key. This redistribution operation involves exchanging rows between parallel execution servers. This is a CPU-intensive operation that can lead to excessive interconnect traffic in Oracle Real Application Clusters environments. Partitioning large tables on a join key, either a foreign or primary key, prevents this redistribution every time the table is joined on that key. Of course, if you choose a foreign key to partition the table, which is the most common scenario, select a foreign key that is involved in many queries.

To illustrate partial partition-wise joins, consider the previous `sales/customer` example. Assume that `customers` is not partitioned or is partitioned on a column other than `cust_id`. Because `sales` is often joined with `customers` on `cust_id`, and because this join dominates our application workload, partition `sales` on `cust_id` to enable partial partition-wise joins every time `customers` and `sales` are joined. As with full partition-wise joins, you have several alternatives:

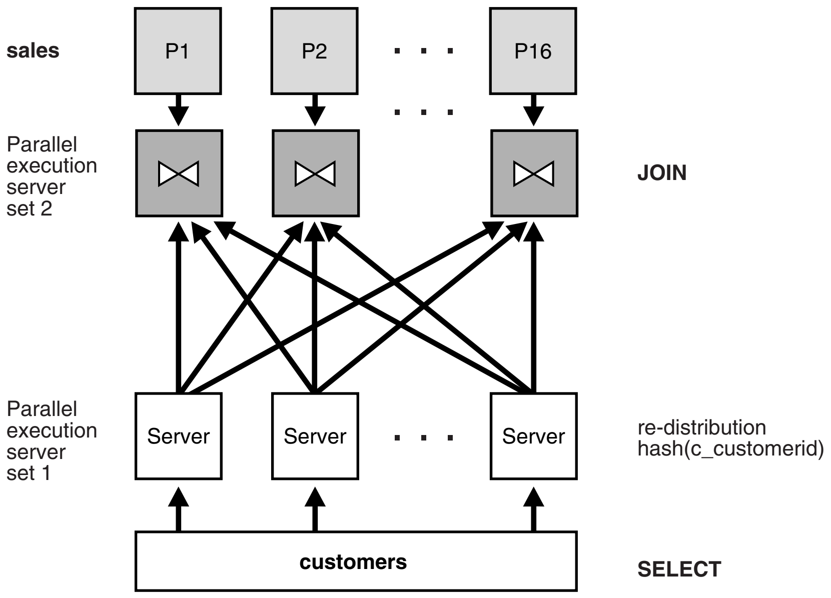
Partial Partition-Wise Joins: Single-Level Partitioning

The simplest method to enable a partial partition-wise join is to partition `sales` by hash on `cust_id`. The number of partitions determines the maximum degree of parallelism, because the partition is the smallest granule of parallelism for partial partition-wise join operations.

The parallel execution of a partial partition-wise join is illustrated in Figure 4–3, which assumes that both the degree of parallelism and the number of partitions of `sales` are 16. The execution involves two sets of query servers: one set, labeled *set 1* in Figure 4–3, scans the `customers` table in parallel. The granule of parallelism for the scan operation is a range of blocks.

Rows from `customers` that are selected by the first set, in this case all rows, are redistributed to the second set of query servers by hashing `cust_id`. For example, all rows in `customers` that could have matching rows in partition `P1` of `sales` are sent to query server 1 in the second set. Rows received by the second set of query servers are joined with the rows from the corresponding partitions in `sales`. Query server number 1 in the second set joins all `customers` rows that it receives with partition `P1` of `sales`.

Figure 4–3 Partial Partition-Wise Join


sales
P1
P2
P16
Parallel execution server set 2
JOIN
Parallel execution server set 1
Server
Server
Server
re-distribution hash(c_customerid)
customers
SELECT

The example below shows the execution plan for the partial partition-wise join between `sales` and `customers`.

```
----------------------------------------------------------------------------------------------
| Id  | Operation                             | Name      | Pstart| Pstop |IN-OUT| PQ Distrib |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |           |       |       |      |            |
|   1 |  PX COORDINATOR                       |           |       |       |      |            |
|   2 |   PX SEND QC (RANDOM)                 | :TQ10002  |       |       | P->S | QC (RAND)  |
|*  3 |    FILTER                             |           |       |       | PCWC |            |
|   4 |     HASH GROUP BY                     |           |       |       | PCWP |            |
|   5 |      PX RECEIVE                       |           |       |       | PCWP |            |
|   6 |       PX SEND HASH                    | :TQ10001  |       |       | P->P | HASH       |
|   7 |        HASH GROUP BY                  |           |       |       | PCWP |            |
|*  8 |         HASH JOIN                     |           |       |       | PCWP |            |
|   9 |          PART JOIN FILTER CREATE      | :BF0000   |       |       | PCWP |            |
|  10 |           PX RECEIVE                  |           |       |       | PCWP |            |
|  11 |            PX SEND PARTITION (KEY)    | :TQ10000  |       |       | P->P | PART (KEY) |
|  12 |             PX BLOCK ITERATOR         |           |       |       | PCWC |            |
|  13 |              TABLE ACCESS FULL        | CUSTOMERS |       |       | PCWP |            |
|  14 |          PX PARTITION HASH JOIN-FILTER|           |:BF0000|:BF0000| PCWC |            |
|* 15 |           TABLE ACCESS FULL           | SALES     |:BF0000|:BF0000| PCWP |            |
----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   8 - access("S"."CUST_ID"="C"."CUST_ID")
  15 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Note: The `Rows`, `Cost (%CPU)`, `Time`, and `TQ` columns were removed from the plan table output in this example.

Note: This section is based on hash partitioning, but it also applies for range, list, and interval partial partition-wise joins.

Considerations for full partition-wise joins also apply to partial partition-wise joins:

- The degree of parallelism does not need to equal the number of partitions. In Figure 4–3, the query executes with two sets of 16 query servers. In this case, Oracle assigns 1 partition to each query server of the second set. Again, the number of partitions should always be a multiple of the degree of parallelism.
- In Oracle Real Application Clusters environments on shared-nothing platforms (MPPs), each hash partition of `sales` should preferably have affinity to only one node in order to avoid remote I/Os. Also, spread partitions over all nodes to avoid bottlenecks and use all CPU resources available on the system. A node can host multiple partitions when there are more partitions than nodes.

See Also: *Oracle Real Application Clusters Administration and Deployment Guide* for more information on data affinity

Partial Partition-Wise Joins: Composite

As with full partition-wise joins, the prime partitioning method for the `sales` table is to use the range method on column `time_id`. This is because `sales` is a typical example of a table that stores historical data. To enable a partial partition-wise join while preserving this range partitioning, subpartition `sales` by hash on column `cust_id` using 16 subpartitions for each partition. Pruning and partial partition-wise joins can be used together if a query joins `customers` and `sales` and if the query has a selection predicate on `time_id`.

When the `sales` table is composite partitioned, the granule of parallelism for a partial partition-wise join is a hash partition and not a subpartition. Refer to Figure 4–2 for an illustration of a hash partition in a composite table. Again, the number of hash partitions should be a multiple of the degree of parallelism. Also, on an MPP system, ensure that each hash partition has affinity to a single node. In the previous example, the eight subpartitions composing a hash partition should have affinity to the same node.

Note: This section is based on range-hash, but it also applies for all other combinations of composite partial partition-wise joins.

The following example shows the execution plan for the query between `sales` and `customers` with sales range partitioned by `time_id` and subpartitioned by hash on `cust_id`.

```
---------------------------------------------------------------------------------------------
| Id  | Operation                            | Name       | Pstart| Pstop |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |            |       |       |      |            |
|   1 |  PX COORDINATOR                      |            |       |       |      |            |
|   2 |   PX SEND QC (RANDOM)                | :TQ10002   |       |       | P->S | QC (RAND)  |
|*  3 |    FILTER                            |            |       |       | PCWC |            |
|   4 |     HASH GROUP BY                    |            |       |       | PCWP |            |
|   5 |      PX RECEIVE                      |            |       |       | PCWP |            |
|   6 |       PX SEND HASH                   | :TQ10001   |       |       | P->P | HASH       |
|   7 |        HASH GROUP BY                 |            |       |       | PCWP |            |
|*  8 |         HASH JOIN                    |            |       |       | PCWP |            |
|   9 |          PART JOIN FILTER CREATE     | :BF0000    |       |       | PCWP |            |
|  10 |           PX RECEIVE                 |            |       |       | PCWP |            |
|  11 |            PX SEND PARTITION (KEY)   | :TQ10000   |       |       | P->P | PART (KEY) |
|  12 |             PX BLOCK ITERATOR        |            |       |       | PCWC |            |
|  13 |              TABLE ACCESS FULL       | CUSTOMERS  |       |       | PCWP |            |
|  14 |          PX PARTITION RANGE ITERATOR|            |     8 |     9 | PCWC |            |
|  15 |           PX PARTITION HASH ALL      |            |     1 |    16 | PCWC |            |
|* 16 |            TABLE ACCESS FULL         | SALES      |   113 |   144 | PCWP |            |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   8 - access("S"."CUST_ID"="C"."CUST_ID")
  16 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Note: The `Rows`, `Cost (%CPU)`, `Time`, and `TQ` columns were removed from the plan table output in this example.

Index Partitioning

The rules for partitioning indexes are similar to those for tables:

- An index can be partitioned unless:
  - The index is a cluster index.
  - The index is defined on a clustered table.
- You can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables:
  - A partitioned table can have partitioned or nonpartitioned indexes.
  - A nonpartitioned table can have partitioned or nonpartitioned indexes.
- Bitmap indexes on nonpartitioned tables cannot be partitioned.
- A bitmap index on a partitioned table must be a local index.

However, partitioned indexes are more complicated than partitioned tables because there are three types of partitioned indexes:

- Local prefixed
- Local nonprefixed
- Global prefixed

These types are described in the following section. Oracle supports all three types.

Local Partitioned Indexes

In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is created by specifying the `LOCAL` attribute.

Oracle constructs the local index so that it is equipartitioned with the underlying table. Oracle partitions the index on the same columns as the underlying table, creates the same number of partitions or subpartitions, and gives them the same partition bounds as corresponding partitions of the underlying table.

Oracle also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced. This ensures that the index remains equipartitioned with the table.

A local index can be created `UNIQUE` if the partitioning columns form a subset of the index columns. This restriction guarantees that rows with identical index keys always map into the same partition, where uniqueness violations can be detected.

Local indexes have the following advantages:

- Only one index partition needs to be rebuilt when a maintenance operation other than `SPLIT PARTITION` or `ADD PARTITION` is performed on an underlying table partition.
- The duration of a partition maintenance operation remains proportional to partition size if the partitioned table has only local indexes.
- Local indexes support partition independence.
- Local indexes support smooth roll-out of old data and roll-in of new data in historical tables.

- Oracle can take advantage of the fact that a local index is equipartitioned with the underlying table to generate better query access plans.
- Local indexes simplify the task of tablespace incomplete recovery. In order to recover a partition or subpartition of a table to a point in time, you must also recover the corresponding index entries to the same point in time. The only way to accomplish this is with a local index. Then you can recover the corresponding table and index partitions or subpartitions together.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for a description of the `DBMS_PCLXUTIL` package

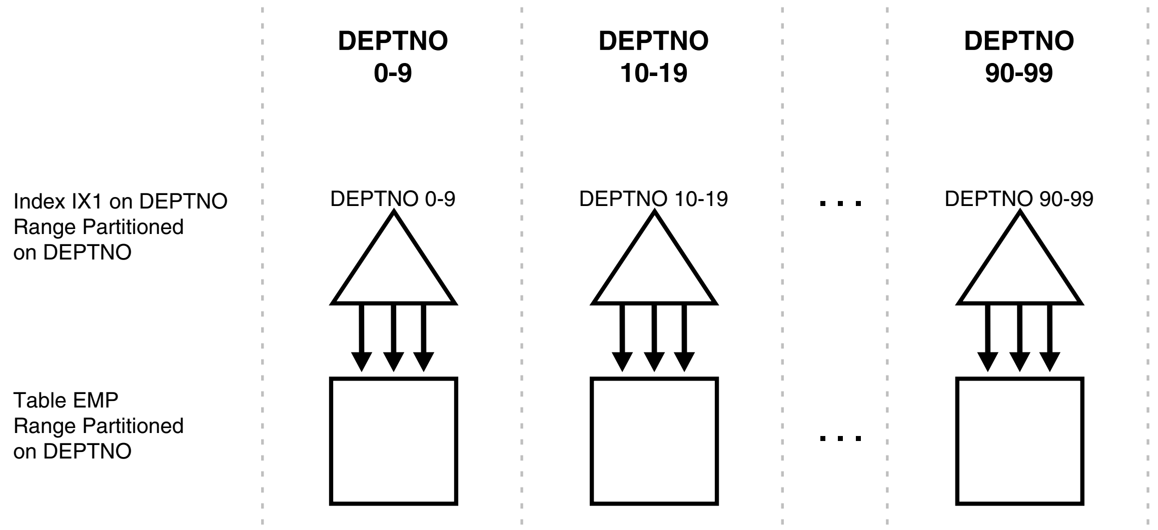
Local Prefixed Indexes

A local index is prefixed if it is partitioned on a left prefix of the index columns. For example, if the `sales` table and its local index `sales_ix` are partitioned on the `week_num` column, then index `sales_ix` is local prefixed if it is defined on the columns (`week_num`, `xaction_num`). On the other hand, if index `sales_ix` is defined on column `product_num` then it is not prefixed.

Local prefixed indexes can be unique or nonunique.

Figure 4–4 illustrates another example of a local prefixed index.

Figure 4–4 Local Prefixed Index


DEPTNO 0-9
DEPTNO 10-19
DEPTNO 90-99
Index IX1 on DEPTNO Range Partitioned on DEPTNO
DEPTNO 0-9
DEPTNO 10-19
. . .
DEPTNO 90-99
Table EMP Range Partitioned on DEPTNO
. . .

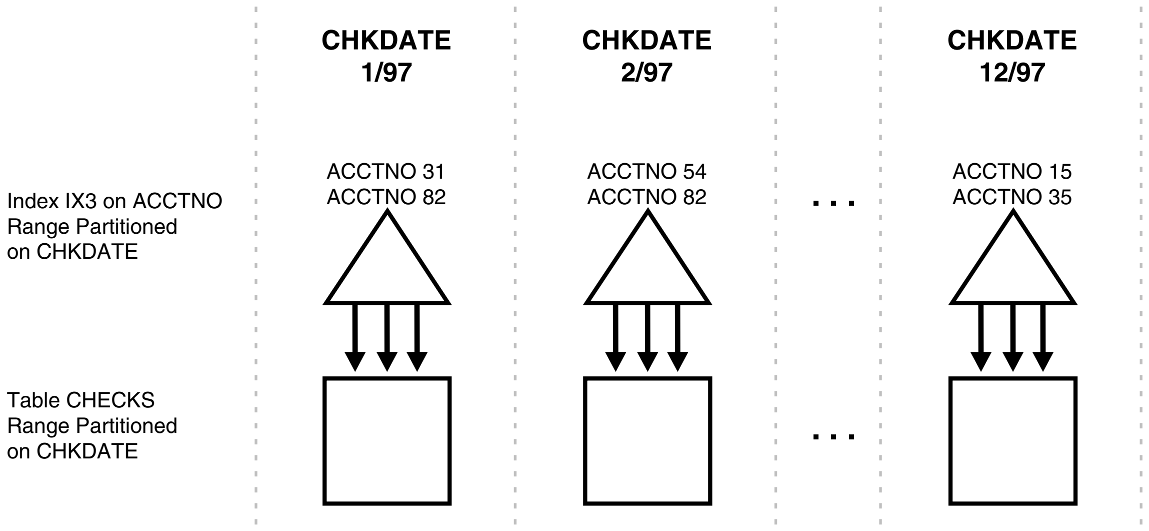
Local Nonprefixed Indexes

A local index is nonprefixed if it is not partitioned on a left prefix of the index columns.

You cannot have a unique local nonprefixed index unless the partitioning key is a subset of the index key.

Figure 4–5 illustrates an example of a local nonprefixed index.

Figure 4–5 Local Nonprefixed Index



Global Partitioned Indexes

In a global partitioned index, the keys in a particular index partition may refer to rows stored in more than one underlying table partition or subpartition. A global index can be range or hash partitioned, though it can be defined on any type of partitioned table.

A global index is created by specifying the `GLOBAL` attribute. The database administrator is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be merged or split as necessary.

Normally, a global index is not equipartitioned with the underlying table. There is nothing to prevent an index from being equipartitioned with the underlying table, but Oracle does not take advantage of the equipartitioning when generating query plans or executing partition maintenance operations. So an index that is equipartitioned with the underlying table should be created as `LOCAL`.

A global partitioned index contains a single B-tree with entries for all rows in all partitions. Each index partition may contain keys that refer to many different partitions or subpartitions in the table.

The highest partition of a global index must have a partition bound all of whose values are `MAXVALUE`. This insures that all rows in the underlying table can be represented in the index.

Prefixed and Nonprefixed Global Partitioned Indexes

A global partitioned index is prefixed if it is partitioned on a left prefix of the index columns. See Figure 4–6 for an example. A global partitioned index is nonprefixed if it is not partitioned on a left prefix of the index columns. Oracle does not support global nonprefixed partitioned indexes.

Global prefixed partitioned indexes can be unique or nonunique.

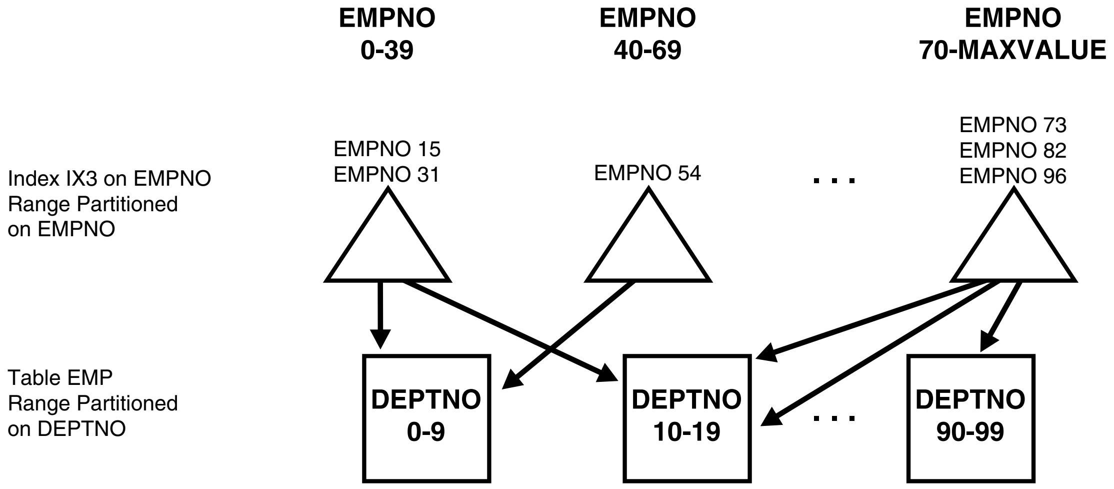
Nonpartitioned indexes are treated as global prefixed nonpartitioned indexes.

Management of Global Partitioned Indexes

Global partitioned indexes are harder to manage than local indexes:

- When the data in an underlying table partition is moved or removed (`SPLIT`, `MOVE`, `DROP`, or `TRUNCATE`), all partitions of a global index are affected. Consequently global indexes do not support partition independence.
- When an underlying table partition or subpartition is recovered to a point in time, all corresponding entries in a global index must be recovered to the same point in time. Because these entries may be scattered across all partitions or subpartitions of the index, mixed in with entries for other partitions or subpartitions that are not being recovered, there is no way to accomplish this except by re-creating the entire global index.

Figure 4–6 Global Prefixed Partitioned Index



Summary of Partitioned Index Types

Table 4–1 summarizes the types of partitioned indexes that Oracle supports. The key points are:

- If an index is local, it is equipartitioned with the underlying table. Otherwise, it is global.
- A prefixed index is partitioned on a left prefix of the index columns. Otherwise, it is nonprefixed.

Table 4–1 Types of Partitioned Indexes

Type of Index	Index Equipartitioned with Table	Index Partitioned on Left Prefix of Index Columns	UNIQUE Attribute Allowed	Example: Table Partitioning Key	Example: Index Columns	Example: Index Partitioning Key
Local Prefixed (any partitioning method)	Yes	Yes	Yes	A	A, B	A
Local Nonprefixed (any partitioning method)	Yes	No	Yes[1]	A	B, A	A
Global Prefixed (range partitioning only)	No[2]	Yes	Yes	A	B	B

[1] For a unique local nonprefixed index, the partitioning key must be a subset of the index key.

[2] Although a global partitioned index may be equipartitioned with the underlying table, Oracle does not take advantage of the partitioning or maintain equipartitioning after partition maintenance operations such as DROP or SPLIT PARTITION.

The Importance of Nonprefixed Indexes

Nonprefixed indexes are particularly useful in historical databases. In a table containing historical data, it is common for an index to be defined on one column to support the requirements of fast access by that column, but partitioned on another column (the same column as the underlying table) to support the time interval for rolling out old data and rolling in new data.

Consider a `sales` table partitioned by week. It contains a year's worth of data, divided into 13 partitions. It is range partitioned on `week_no`, four weeks to a partition. You might create a nonprefixed local index `sales_ix` on `sales`. The `sales_ix` index is defined on `acct_no` because there are queries that need fast access to the data by account number. However, it is partitioned on `week_no` to match the `sales` table. Every four weeks, the oldest partitions of `sales` and `sales_ix` are dropped and new ones are added.

Performance Implications of Prefixed and Nonprefixed Indexes

It is more expensive to probe into a nonprefixed index than to probe into a prefixed index.

If an index is prefixed (either local or global) and Oracle is presented with a predicate involving the index columns, then partition pruning can restrict application of the predicate to a subset of the index partitions.

For example, in Figure 4–4 on page 4-20, if the predicate is `deptno=15`, the optimizer knows to apply the predicate only to the second partition of the index. (If the predicate involves a bind variable, the optimizer will not know exactly which partition but it may still know there is only one partition involved, in which case at run time, only one index partition will be accessed.)

When an index is nonprefixed, Oracle often has to apply a predicate involving the index columns to all `N` index partitions. This is required to look up a single key, or to do an index range scan. For a range scan, Oracle must also combine information from `N` index partitions. For example, in Figure 4–5 on page 4-21, a local index is partitioned on `chkdate` with an index key on `acctno`. If the predicate is `acctno=31`, Oracle probes all 12 index partitions.

Of course, if there is also a predicate on the partitioning columns, then multiple index probes might not be necessary. Oracle takes advantage of the fact that a local index is equipartitioned with the underlying table to prune partitions based on the partition key. For example, if the predicate in Figure 4–4 on page 4-20 is `chkdate<3/97`, Oracle only has to probe two partitions.

So for a nonprefixed index, if the partition key is a part of the `WHERE` clause but not of the index key, then the optimizer determines which index partitions to probe based on the underlying table partition.

When many queries and DML statements using keys of local, nonprefixed, indexes have to probe all index partitions, this effectively reduces the degree of partition independence provided by such indexes.

Table 4–2 Comparing Prefixed Local, Nonprefixed Local, and Global Indexes

Index Characteristics	Prefixed Local	Nonprefixed Local	Global
Unique possible?	Yes	Yes	Yes. Must be global if using indexes on columns other than the partitioning columns

Table 4–2 (Cont.) Comparing Prefixed Local, Nonprefixed Local, and Global Indexes

Index Characteristics	Prefixed Local	Nonprefixed Local	Global
Manageability	Easy to manage	Easy to manage	Harder to manage
OLTP	Good	Bad	Good
Long Running (DSS)	Good	Good	Not Good

Guidelines for Partitioning Indexes

When deciding how to partition indexes on a table, consider the mix of applications that need to access the table. There is a trade-off between performance on the one hand and availability and manageability on the other. Here are some of the guidelines you should consider:

- For OLTP applications:
  - Global indexes and local prefixed indexes provide better performance than local nonprefixed indexes because they minimize the number of index partition probes.
  - Local indexes support more availability when there are partition or subpartition maintenance operations on the table. Local nonprefixed indexes are very useful for historical databases.
- For DSS applications, local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

  For example, a query using the predicate "`acctno` between 40 and 45" on the table `checks` of Figure 4–4 on page 4-20 causes parallel scans of all the partitions of the nonprefixed index `ix3`. On the other hand, a query using the predicate `deptno BETWEEN 40 AND 45` on the table `deptno` of Figure 4–5 on page 4-21 cannot be parallelized because it accesses a single partition of the prefixed index `ix1`.
- For historical tables, indexes should be local if possible. This limits the impact of regularly scheduled drop partition operations.
- Unique indexes on columns other than the partitioning columns must be global because unique local nonprefixed indexes whose key does not contain the partitioning key are not supported.

Physical Attributes of Index Partitions

Default physical attributes are initially specified when a `CREATE INDEX` statement creates a partitioned index. Because there is no segment corresponding to the partitioned index itself, these attributes are only used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using `ALTER INDEX MODIFY DEFAULT ATTRIBUTES`.

Physical attributes of partitions created by `CREATE INDEX` are determined as follows:

- Values of physical attributes specified (explicitly or by default) for the index are used whenever the value of a corresponding partition attribute is not specified. Handling of the `TABLESPACE` attribute of partitions of a `LOCAL` index constitutes an important exception to this rule in that in the absence of a user-specified `TABLESPACE` value (at both partition and index levels), that of the corresponding partition of the underlying table is used.

- Physical attributes (other than `TABLESPACE`, as explained in the preceding) of partitions of local indexes created in the course of processing `ALTER TABLE ADD PARTITION` are set to the default physical attributes of each index.

Physical attributes (other than `TABLESPACE`) of index partitions created by `ALTER TABLE SPLIT PARTITION` are determined as follows:

- Values of physical attributes of the index partition being split are used.

Physical attributes of an existing index partition can be modified by `ALTER INDEX MODIFY PARTITION` and `ALTER INDEX REBUILD PARTITION`. Resulting attributes are determined as follows:

- Values of physical attributes of the partition before the statement was issued are used whenever a new value is not specified. Note that `ALTER INDEX REBUILD PARTITION` can be used to change the tablespace in which a partition resides.

Physical attributes of global index partitions created by `ALTER INDEX SPLIT PARTITION` are determined as follows:

- Values of physical attributes of the partition being split are used whenever a new value is not specified.
- Physical attributes of all partitions of an index (along with default values) may be modified by `ALTER INDEX`, for example, `ALTER INDEX indexname NOLOGGING` changes the logging mode of all partitions of `indexname` to `NOLOGGING`.

See Also: Chapter 3, "Partition Administration" for more detailed examples of adding partitions and examples of rebuilding indexes

Partitioning and Table Compression

You can compress several partitions or a complete partitioned heap-organized table. You do this by either 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 table level, from the tablespace definition.

To decide whether or not a partition should be compressed or stay uncompressed adheres to the same rules as a nonpartitioned table. However, due to the capability of range and composite partitioning to separate data logically into distinct partitions, such 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 segment compression, you can keep more old data online, minimizing the burden of additional storage consumption.

You can also change any existing uncompressed table partition later on, 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 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. Everything that applies to fully uncompressed partitioned tables is also valid for partially or fully compressed partitioned tables.

See Also: *Oracle Database Data Warehousing Guide* for a generic discussion of table compression and *Oracle Database Performance Tuning Guide* for an example of calculating the compression ratio

Table Compression and Bitmap Indexes

If you want to use table compression on partitioned tables with bitmap indexes, you need to do the following before you introduce the compression attribute for the first time:

1. Mark bitmap indexes unusable.
2. Set the compression attribute.
3. Rebuild the indexes.

The first time you make a compressed partition part of an already existing, fully uncompressed partitioned table, you must either drop all existing bitmap indexes or mark them `UNUSABLE` prior to adding a compressed partition. This must be done irrespective of whether any partition contains any data. It is also independent of the operation that causes one or more compressed partitions to become part of the table. This does not apply to a partitioned table having B-tree indexes only.

This rebuilding of the bitmap index structures is necessary to accommodate the potentially higher number of rows stored for each data block with table compression enabled and must be done only for the first time. All subsequent operations, whether they affect compressed or uncompressed partitions, or change the compression attribute, behave identically for uncompressed, partially compressed, or fully compressed partitioned tables.

To avoid the recreation of any bitmap index structure, Oracle recommends creating every partitioned table with at least one compressed partition whenever you plan to partially or fully compress the partitioned table in the future. This compressed partition can stay empty or even can be dropped after the partition table creation.

Having a partitioned table with compressed partitions can lead to slightly larger bitmap index structures for the uncompressed partitions. The bitmap index structures for the compressed partitions, however, are in most cases smaller than the appropriate bitmap index structure before table compression. This highly depends on the achieved compression rates.

Note: Oracle Database will raise an error if compression is introduced to an object for the first time and there are usable bitmap index segments.

Example of Table Compression and Partitioning

The following statement moves and compresses an already existing partition `sales_q1_1998` of table `sales`:

```
ALTER TABLE sales
MOVE PARTITION sales_q1_1998 TABLESPACE ts_arch_q1_1998 COMPRESS;
```

If you use the `MOVE` statement, the local indexes for partition `sales_q1_1998` become unusable. You have to rebuild them afterward, as follows:

```
ALTER TABLE sales
MODIFY PARTITION sales_q1_1998 REBUILD UNUSABLE LOCAL INDEXES;
```

You can also include the `UPDATE INDEXES` clause in the `MOVE` statement in order for the entire operation to be completed automatically without any negative impact on users accessing the table.

The following statement merges two existing partitions into a new, compressed partition, residing in a separate tablespace. The local bitmap indexes have to be rebuilt afterward, as follows:

```
ALTER TABLE sales MERGE PARTITIONS sales_q1_1998, sales_q2_1998
INTO PARTITION sales_1_1998 TABLESPACE ts_arch_1_1998
COMPRESS FOR ALL OPERATIONS UPDATE INDEXES;
```

See Also:

- Chapter 3, "Partition Administration" for more details and examples for the partition management operations
- *Oracle Database Performance Tuning Guide* for details regarding how to estimate the compression ratio when using table compression

Recommendations for Choosing a Partitioning Strategy

The following sections provide recommendations for choosing a partitioning strategy:

- When to Use Range or Interval Partitioning
- When to Use Hash Partitioning
- When to Use List Partitioning
- When to Use Composite Partitioning
- When to Use Interval Partitioning
- When to Use Reference Partitioning
- When to Partition on Virtual Columns

When to Use Range or Interval Partitioning

Range partitioning is a convenient method for partitioning historical data. The boundaries of range partitions define the ordering of the partitions in the tables or indexes.

Interval partitioning is an extension to range partitioning in which, beyond a point in time, partitions are defined by an interval. Interval partitions are automatically created by the database when data is inserted into the partition.

Range or interval partitioning is often used to organize data by time intervals on a column of type `DATE`. Thus, most SQL statements accessing range partitions focus on timeframes. An example of this is a SQL statement similar to "select data from a particular period in time." In such a scenario, if each partition represents data for one month, the query "find data of month 06-DEC" needs to access only the December partition of year 2006. This reduces the amount of data scanned to a fraction of the total data available, an optimization method called partition pruning.

Range partitioning is also ideal when you periodically load new data and purge old data, because it is easy to add or drop partitions.

It is common to keep a rolling window of data, for example keeping the past 36 months' worth of data online. Range partitioning simplifies this process. To add data from a new month, you load it into a separate table, clean it, index it, and then add it to the range-partitioned table using the `EXCHANGE PARTITION` statement, all while the original table remains online. Once you add the new partition, you can drop the trailing month with the `DROP PARTITION` statement. The alternative to using the `DROP PARTITION` statement can be to archive the partition and make it read only, but this works only when your partitions are in separate tablespaces. You can also implement a rolling window data using inserts into the partitioned table.

Interval partitioning provides an easy way for interval partitions to be automatically created as data arrives. Interval partitions can also be used for all other partition maintenance operations. Refer to Chapter 3, "Partition Administration" for more details on the partition maintenance operations on interval partitions.

In conclusion, consider using range or interval partitioning when:

- Very large tables are frequently scanned by a range predicate on a good partitioning column, such as `ORDER_DATE` or `PURCHASE_DATE`. Partitioning the table on that column enables partition pruning.
- You want to maintain a rolling window of data.
- You cannot complete administrative operations, such as backup and restore, on large tables in an allotted time frame, but you can divide them into smaller logical pieces based on the partition range column.

The following example creates the table `salestable` for a period of two years, 2005 and 2006, and partitions it by range according to the column `s_salesdate` to separate the data into eight quarters, each corresponding to a partition. Future partitions are created automatically through the monthly interval definition. Interval partitions are created in the provided list of tablespaces in a round-robin manner. Analysis of sales figures by a period of time can take advantage of partition pruning. The sales table also supports a rolling window approach.

```
CREATE TABLE salestable
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY RANGE(s_saledate)
INTERVAL(NUMTOYMINTERVAL(1,'MONTH')) STORE IN (tbs1,tbs2,tbs3,tbs4)
 (PARTITION sal05q1 VALUES LESS THAN (TO_DATE('01-APR-2005', 'DD-MON-YYYY'))
   TABLESPACE tbs1,
  PARTITION sal05q2 VALUES LESS THAN (TO_DATE('01-JUL-2005', 'DD-MON-YYYY'))
   TABLESPACE tbs2,
  PARTITION sal05q3 VALUES LESS THAN (TO_DATE('01-OCT-2005', 'DD-MON-YYYY'))
   TABLESPACE tbs3,
  PARTITION sal05q4 VALUES LESS THAN (TO_DATE('01-JAN-2006', 'DD-MON-YYYY'))
   TABLESPACE tbs4,
  PARTITION sal06q1 VALUES LESS THAN (TO_DATE('01-APR-2006', 'DD-MON-YYYY'))
   TABLESPACE tbs1,
  PARTITION sal06q2 VALUES LESS THAN (TO_DATE('01-JUL-2006', 'DD-MON-YYYY'))
   TABLESPACE tbs2,
  PARTITION sal06q3 VALUES LESS THAN (TO_DATE('01-OCT-2006', 'DD-MON-YYYY'))
   TABLESPACE tbs3,
  PARTITION sal06q4 VALUES LESS THAN (TO_DATE('01-JAN-2007', 'DD-MON-YYYY'))
   TABLESPACE tbs4);
```

When to Use Hash Partitioning

There are times when it is not obvious into which partition data should reside, although the partitioning key can be identified. Rather than group similar data together, there are times when it is desirable to distribute data such that it does not correspond to a business or a logical view of the data, as it does in range partitioning. With hash partitioning, a row is placed into a partition based on the result of passing the partitioning key into a hashing algorithm.

Using this approach, data is randomly distributed across the partitions rather than grouped together. Hence, this is a good approach for some data, but may not be an effective way to manage historical data. However, hash partitions share some performance characteristics with range partitions. For example, partition pruning is limited to equality predicates. You can also use partition-wise joins, parallel index access, and parallel DML. See "Partition-Wise Joins" on page 4-10 for more information.

As a general rule, use hash partitioning for the following purposes:

- To enable partial or full parallel partition-wise joins with very likely equi-sized partitions.
- To distribute data evenly among the nodes of an MPP platform that uses Oracle Real Application Clusters. As a result, you can minimize interconnect traffic when processing internode parallel statements.
- To use partition pruning and partition-wise joins according to a partitioning key that is mostly constrained by a distinct value or value list.
- To randomly distribute data to avoid I/O bottlenecks if you do not use a storage management technique that stripes and mirrors across all available devices.

See Also: Chapter 9, "Storage Management for VLDBs" for more information

Note: With hash partitioning, only equality or `IN`-list predicates are supported for partition pruning.

For optimal data distribution, the following requirements should be satisfied:

- Choose a column or combination of columns that is unique or almost unique.
- Create a number of partitions and subpartitions for each partition that is a power of two. For example, 2, 4, 8, 16, 32, 64, 128, and so on.

The following example creates four hash partitions for the table `sales_hash` using the column `s_productid` as the partitioning key. Parallel joins with the products table can take advantage of partial or full partition-wise joins. Queries accessing sales figures for only a single product or a list of products will benefit from partition pruning.

```
CREATE TABLE sales_hash
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY HASH(s_productid)
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
);
```

If you do not explicitly specify partition names, but rather the number of hash partitions, then Oracle automatically generates internal names for the partitions. Also, you can use the `STORE IN` clause to assign hash partitions to tablespaces in a round-robin manner.

See Also:

- *Oracle Database SQL Language Reference* for partitioning syntax
- Chapter 3, "Partition Administration" for more examples

When to Use List Partitioning

You should use list partitioning when you want to specifically map rows to partitions based on discrete values. For example, all the customers for states Oregon and Washington are stored in one partition and customers in other states are stored in different partitions. Account managers who analyze their accounts by region can take advantage of partition pruning.

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
( PARTITION p_northwest VALUES ('OR', 'WA')
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
, PARTITION p_southeast VALUES ('FL', 'GA')
, PARTITION p_northcentral VALUES ('SD', 'WI')
, PARTITION p_southcentral VALUES ('OK', 'TX')
);
```

Unlike range and hash partitioning, multi-column partition keys are not supported for list partitioning. If a table is partitioned by list, the partitioning key can only consist of a single column of the table.

When to Use Composite Partitioning

Composite partitioning offers the benefits of partitioning on two dimensions. From a performance perspective you can take advantage of partition pruning on one or two dimensions depending on the SQL statement, and you can take advantage of the use of full or partial partition-wise joins on either dimension.

You can take advantage of parallel backup and recovery of a single table. Composite partitioning also increases the number of partitions significantly, which may be beneficial for efficient parallel execution. From a manageability perspective, you can implement a rolling window to support historical data and still partition on another dimension if many statements can benefit from partition pruning or partition-wise joins.

You can split up backups of your tables and you can decide to store data differently based on identification by a partitioning key. For example, you may decide to store data for a specific product type in a read-only, compressed format, and keep other product type data uncompressed.

The database stores every subpartition in a composite partitioned table as a separate segment. Thus, the subpartitions may have properties that differ from the properties of the table or from the partition to which the subpartitions belong.

> **See Also:** *Oracle Database SQL Language Reference* for details regarding syntax and restrictions

When to Use Composite Range-Hash Partitioning

Composite range-hash partitioning is particularly common for tables that store history, are very large as a result, and are frequently joined with other large tables. For these types of tables (typical of data warehouse systems), composite range-hash partitioning provides the benefit of partition pruning at the range level with the opportunity to perform parallel full or partial partition-wise joins at the hash level. Specific cases can benefit from partition pruning on both dimensions for specific SQL statements.

Composite range-hash partitioning can also be used for tables that traditionally use hash partitioning, but also use a rolling window approach. Over time, data can be moved from one storage tier to another storage tier, compressed, stored in a read-only tablespace, and eventually purged. Information Lifecycle Management (ILM) scenarios often use range partitions to implement a tiered storage approach. See Chapter 5, "Using Partitioning for Information Lifecycle Management" for more details.

The following is an example of a range-hash partitioned `page_history` table of an Internet service provider. The table definition is optimized for historical analysis for either specific `client_ip` values (in which case queries benefit from partition pruning) or for analysis across many IP addresses, in which case queries can take advantage of full or partial partition-wise joins.

```
CREATE TABLE page_history
( id                NUMBER NOT NULL
, url               VARCHAR2(300) NOT NULL
, view_date         DATE NOT NULL
, client_ip         VARCHAR2(23) NOT NULL
, from_url          VARCHAR2(300)
, to_url            VARCHAR2(300)
, timing_in_seconds NUMBER
) PARTITION BY RANGE(view_date) INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY HASH(client_ip)
SUBPARTITIONS 32
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2006','dd-MON-yyyy')))
PARALLEL 32 COMPRESS;
```

This example shows the use of interval partitioning. Interval partitioning can be used in addition to range partitioning in order for interval partitions to be created automatically as data is inserted into the table.

When to Use Composite Range-List Partitioning

Composite range-list partitioning is commonly used for large tables that store historical data and are commonly accessed on more than one dimension. Often the historical view of the data is one access path, but certain business cases add another categorization to the access path. For example, regional account managers are very interested in how many new customers they signed up in their region in a specific time period. ILM and its tiered storage approach is a common reason to create range-list partitioned tables so that older data can be moved and compressed, but partition pruning on the list dimension is still available.

The following example creates a range-list partitioned `call_detail_records` table. A telecom company can use this table to analyze specific types of calls over time. The table uses local indexes on `from_number` and `to_number`.

```
CREATE TABLE call_detail_records
( id NUMBER
, from_number        VARCHAR2(20)
, to_number          VARCHAR2(20)
, date_of_call       DATE
, distance           VARCHAR2(1)
, call_duration_in_s NUMBER(4)
) PARTITION BY RANGE(date_of_call)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY LIST(distance)
SUBPARTITION TEMPLATE
( SUBPARTITION local VALUES('L') TABLESPACE tbs1
, SUBPARTITION medium_long VALUES ('M') TABLESPACE tbs2
, SUBPARTITION long_distance VALUES ('D') TABLESPACE tbs3
, SUBPARTITION international VALUES ('I') TABLESPACE tbs4
)
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2005','dd-MON-yyyy')))
PARALLEL;

CREATE INDEX from_number_ix ON call_detail_records(from_number)
LOCAL PARALLEL NOLOGGING;

CREATE INDEX to_number_ix ON call_detail_records(to_number)
LOCAL PARALLEL NOLOGGING;
```

This example shows the use of interval partitioning. Interval partitioning can be used in addition to range partitioning in order for interval partitions to be created automatically as data is inserted into the table.

When to Use Composite Range-Range Partitioning

Composite range-range partitioning is useful for applications that store time-dependent data on more than one time dimension. Often these applications do not use one particular time dimension to access the data, but rather another time dimension, or sometimes both at the same time. For example, a web retailer wants to analyze its sales data based on when orders were placed, and when orders were shipped (handed over to the shipping company).

Other business cases for composite range-range partitioning include ILM scenarios, and applications that store historical data and want to categorize its data by range on another dimension.

The following example shows a range-range partitioned table `account_balance_history`. A bank may use access to individual subpartitions to contact its customers for low-balance reminders or specific promotions relevant to a certain category of customers.

```
CREATE TABLE account_balance_history
( id                 NUMBER NOT NULL
, account_number     NUMBER NOT NULL
, customer_id        NUMBER NOT NULL
, transaction_date   DATE NOT NULL
, amount_credited    NUMBER
, amount_debited     NUMBER
, end_of_day_balance NUMBER NOT NULL
) PARTITION BY RANGE(transaction_date)
INTERVAL (NUMTODSINTERVAL(7,'DAY'))
```

```
SUBPARTITION BY RANGE(end_of_day_balance)
SUBPARTITION TEMPLATE
( SUBPARTITION unacceptable VALUES LESS THAN (-1000)
, SUBPARTITION credit VALUES LESS THAN (0)
, SUBPARTITION low VALUES LESS THAN (500)
, SUBPARTITION normal VALUES LESS THAN (5000)
, SUBPARTITION high VALUES LESS THAN (20000)
, SUBPARTITION extraordinary VALUES LESS THAN (MAXVALUE)
)
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy')));
```

This example shows the use of interval partitioning. Interval partitioning can be used in addition to range partitioning in order for interval partitions to be created automatically as data is inserted into the table. In this case 7-day (weekly) intervals are created, starting Monday, January 1, 2007.

When to Use Composite List-Hash Partitioning

Composite list-hash partitioning is useful for large tables that are usually accessed on one dimension, but (due to their size) still need to take advantage of parallel full or partial partition-wise joins on another dimension in joins with other large tables.

The following example shows a `credit_card_accounts` table. The table is list-partitioned on region in order for account managers to quickly access accounts in their region. The subpartitioning strategy is hash on `customer_id` so that queries against the transactions table, also subpartitioned on `customer_id`, can take advantage of full partition-wise joins. Joins with the hash-partitioned customers table can also benefit from full partition-wise joins. The table has a local bitmap index on the `is_active` column.

```
CREATE TABLE credit_card_accounts
( account_number  NUMBER(16) NOT NULL
, customer_id     NUMBER NOT NULL
, customer_region VARCHAR2(2) NOT NULL
, is_active       VARCHAR2(1) NOT NULL
, date_opened     DATE NOT NULL
) PARTITION BY LIST (customer_region)
SUBPARTITION BY HASH (customer_id)
SUBPARTITIONS 16
( PARTITION emea VALUES ('EU','ME','AF')
, PARTITION amer VALUES ('NA','LA')
, PARTITION apac VALUES ('SA','AU','NZ','IN','CH')
) PARALLEL;

CREATE BITMAP INDEX is_active_bix ON credit_card_accounts(is_active)
LOCAL PARALLEL NOLOGGING;
```

When to Use Composite List-List Partitioning

Composite list-list partitioning is useful for large tables that are often accessed on different dimensions. You can specifically map rows to partitions on those dimensions based on discrete values.

The following example shows an example of a very frequently accessed `current_inventory` table. The table is constantly updated with the current inventory in the supermarket supplier's local warehouses. Potentially perishable foods are supplied from those warehouses to supermarkets, and it is important to optimize supplies and deliveries. The table has local indexes on `warehouse_id` and `product_id`.

```
CREATE TABLE current_inventory
( warehouse_id      NUMBER
, warehouse_region  VARCHAR2(2)
, product_id        NUMBER
, product_category  VARCHAR2(12)
, amount_in_stock   NUMBER
, unit_of_shipping  VARCHAR2(20)
, products_per_unit NUMBER
, last_updated      DATE
) PARTITION BY LIST (warehouse_region)
SUBPARTITION BY LIST (product_category)
SUBPARTITION TEMPLATE
( SUBPARTITION perishable VALUES ('DAIRY','PRODUCE','MEAT','BREAD')
, SUBPARTITION non_perishable VALUES ('CANNED','PACKAGED')
, SUBPARTITION durable VALUES ('TOYS','KITCHENWARE')
)
( PARTITION p_northwest VALUES ('OR', 'WA')
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
, PARTITION p_southeast VALUES ('FL', 'GA')
, PARTITION p_northcentral VALUES ('SD', 'WI')
, PARTITION p_southcentral VALUES ('OK', 'TX')
);

CREATE INDEX warehouse_id_ix ON current_inventory(warehouse_id)
LOCAL PARALLEL NOLOGGING;

CREATE INDEX product_id_ix ON current_inventory(product_id)
LOCAL PARALLEL NOLOGGING;
```

When to Use Composite List-Range Partitioning

Composite list-range partitioning is useful for large tables that are accessed on different dimensions. For the most commonly used dimension, you can specifically map rows to partitions on discrete values. List-range partitioning is commonly used for tables that use range values within a list partition, whereas range-list partitioning is commonly used for discrete list values within a range partition. List-range partitioning is less commonly used to store historical data, even though equivalent scenarios all work. Range-list partitioning can be implemented using interval-list partitioning, whereas list-range partitioning does not support interval partitioning.

The following example shows a `donations` table that stores donations in different currencies. The donations are categorized into small, medium, and high, depending on the amount. Due to currency differences, the ranges are different.

```
CREATE TABLE donations
( id             NUMBER
, name           VARCHAR2(60)
, beneficiary    VARCHAR2(80)
, payment_method VARCHAR2(30)
, currency       VARCHAR2(3)
, amount         NUMBER
) PARTITION BY LIST (currency)
SUBPARTITION BY RANGE (amount)
( PARTITION p_eur VALUES ('EUR')
  ( SUBPARTITION p_eur_small VALUES LESS THAN (8)
  , SUBPARTITION p_eur_medium VALUES LESS THAN (80)
  , SUBPARTITION p_eur_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_gbp VALUES ('GBP')
  ( SUBPARTITION p_gbp_small VALUES LESS THAN (5)
  , SUBPARTITION p_gbp_medium VALUES LESS THAN (50)
  , SUBPARTITION p_gbp_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_aud_nzd_chf VALUES ('AUD','NZD','CHF')
  ( SUBPARTITION p_aud_nzd_chf_small VALUES LESS THAN (12)
  , SUBPARTITION p_aud_nzd_chf_medium VALUES LESS THAN (120)
  , SUBPARTITION p_aud_nzd_chf_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_jpy VALUES ('JPY')
  ( SUBPARTITION p_jpy_small VALUES LESS THAN (1200)
  , SUBPARTITION p_jpy_medium VALUES LESS THAN (12000)
  , SUBPARTITION p_jpy_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_inr VALUES ('INR')
  ( SUBPARTITION p_inr_small VALUES LESS THAN (400)
  , SUBPARTITION p_inr_medium VALUES LESS THAN (4000)
  , SUBPARTITION p_inr_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_zar VALUES ('ZAR')
  ( SUBPARTITION p_zar_small VALUES LESS THAN (70)
  , SUBPARTITION p_zar_medium VALUES LESS THAN (700)
  , SUBPARTITION p_zar_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_default VALUES (DEFAULT)
  ( SUBPARTITION p_default_small VALUES LESS THAN (10)
  , SUBPARTITION p_default_medium VALUES LESS THAN (100)
  , SUBPARTITION p_default_high VALUES LESS THAN (MAXVALUE)
  )
) ENABLE ROW MOVEMENT;
```

When to Use Interval Partitioning

Interval partitioning can be used for every table that is range partitioned and uses fixed intervals for new partitions. The database automatically creates interval partitions as data for that partition arrives. Until this happens, the interval partition exists but no segment is created for the partition.

The benefit of interval partitioning is that you do not need to create your range partitions explicitly. You should consider using interval partitioning unless you create range partitions with different intervals, or if you always set specific partition attributes when you create range partitions. Note that you can specify a list of tablespaces in the interval definition. The database will create interval partitions in the provided list of tablespaces in a round-robin manner.

If you upgrade your application and you use range partitioning or composite range-* partitioning, then you can easily change your existing table definition to use interval partitioning. Note that you cannot manually add partitions to an interval-partitioned table. If you have automated the creation of new partitions, then you have to change your application code to prevent the explicit creation of range partitions going forward.

The following example shows how to change the `sales` table in the sample `sh` schema from range partitioning to start using monthly interval partitioning.

```
ALTER TABLE sales SET INTERVAL (NUMTOYMINTERVAL(1,'MONTH'));
```

You cannot use interval partitioning with reference partitioned tables.

When to Use Reference Partitioning

Reference partitioning is useful in the following scenarios:

- If you have denormalized, or would denormalize, a column from a master table into a child table in order to get partition pruning benefits on both tables.

  For example, your `orders` table stores the `order_date`, but the `order_items` table, which stores one or more items per order, does not. In order to get good performance for historical analysis of orders data, you would traditionally duplicate the `order_date` column in the `order_items` table to get partition pruning on the `order_items` table.

  You should consider reference partitioning in such a scenario and avoid having to duplicate the `order_date` column. Queries that join both tables and use a predicate on `order_date` automatically benefit from partition pruning on both tables.

- If two large tables are joined frequently, then the tables are not partitioned on the join key, but you want to take advantage of partition-wise joins.

  Reference partitioning implicitly enables full partition-wise joins.

- If data in multiple tables has a related life cycle, then reference partitioning can provide significant manageability benefits.

  Partition management operations against the master table are automatically cascaded to its descendents. For example, when you add a partition to the master table, that creation is automatically propagated to all its descendents.

  In order to use reference partitioning, you have to enable and enforce the foreign key relationship between the master table and the reference table in place. You can cascade reference-partitioned tables.

When to Partition on Virtual Columns

Virtual column partitioning enables you to partition on an expression, which may use data from other columns, and perform calculations with these columns. PL/SQL function calls are not supported in virtual column definitions that are to be used as a partitioning key.

Virtual column partitioning supports all partitioning methods as well as performance and manageability features. You should consider using virtual columns if tables are frequently accessed using a predicate that is not directly captured in a column, but can be derived, in order to get partition pruning benefits. Traditionally, in order to get partition pruning benefits, you would have to add a separate column in order to capture and calculate the correct value and make sure the column is always populated correctly in order to ensure correct query retrieval.

The following example shows a `car_rentals` table. The customer's confirmation number contains a two-character country name as the location where the rental car is picked up. Rental car analyses usually evaluate regional patterns, so it makes sense to partition by country.

```
CREATE TABLE car_rentals
( id                  NUMBER NOT NULL
, customer_id         NUMBER NOT NULL
, confirmation_number VARCHAR2(12) NOT NULL
, car_id              NUMBER
, car_type            VARCHAR2(10)
, requested_car_type  VARCHAR2(10) NOT NULL
, reservation_date    DATE NOT NULL
, start_date          DATE NOT NULL
, end_date            DATE
, country as (substr(confirmation_number,9,2))
) PARTITION BY LIST (country)
SUBPARTITION BY HASH (customer_id)
SUBPARTITIONS 16
( PARTITION north_america VALUES ('US','CA','MX')
, PARTITION south_america VALUES ('BR','AR','PE')
, PARTITION europe VALUES ('GB','DE','NL','BE','FR','ES','IT','CH')
, PARTITION apac VALUES ('NZ','AU','IN','CN')
) ENABLE ROW MOVEMENT;
```

In this example, the column `country` is defined as a virtual column derived from the confirmation number. The virtual column does not require any storage. As the example illustrates, row movement is supported with virtual columns. The database will migrate a row to a different partition if the virtual column evaluates to a different value in another partition.

5

Using Partitioning for Information Lifecycle Management

Although most organizations have long regarded their stores of data as one of their most valuable corporate assets, how this data was managed and maintained varies enormously. Originally, data was used to help achieve operational goals, run the business, and help identify the future direction and success of the company.

However, new government regulations and guidelines are a key driving force in how and why data is being retained, as they are now requiring organizations to retain and control information for very long periods of time. Consequently, today there are two additional objectives IT managers are trying to satisfy: to store vast quantities of data, for the lowest possible cost; and to meet the new regulatory requirements for data retention and protection.

This chapter discusses the components in the Oracle Database which can be used to build an Information Lifecycle Management (ILM) strategy. This chapter contains the following topics:

What Is ILM?

Information today comes in a wide variety of types, for example an E-mail message, a photograph, or an order in an Online Transaction Processing System. Therefore, once you know the type of data and how it will be used, you already have an understanding of what its evolution and final destiny is likely to be.

One of the challenges facing each organization is to understand how its data evolves and grows, monitor how its usage changes over time, and decide how long it should survive, while adhering to all the rules and regulations that now apply to that data. Information Lifecycle Management (ILM) is designed to address these issues, with a combination of processes, policies, software, and hardware so that the appropriate technology can be used for each stage in the lifecycle of the data.

Oracle Database for ILM

The Oracle Database provides the ideal platform for implementing an ILM solution, because it offers:

- Application Transparency

  Application Transparency is very important in ILM because it means that there is no need to customize applications and it also allows various changes to be made to the data without any impact on the applications that are using that data. Therefore, data can easily be moved at the different stages of its lifecycle and access to the data can be optimized via the database. Another important benefit is that application transparency offers the flexibility required to quickly adapt to any new regulatory requirements, again without any impact on the existing applications.

- Fine-grained

  Oracle is able to view data at a very fine-grained level as well as group related data together, whereas storage devices only see bytes and blocks.

- Low-Cost

  With so much data to retain, using low cost storage is a key factor in implementing ILM. Since Oracle can take advantage of many types of storage devices, the maximum amount of data can be held for the lowest possible cost.

- Enforceable Compliance Policies

  When information is kept for compliance reasons, it is imperative to show to regulatory bodies that data is being retained and managed in accordance with the regulations. Within Oracle, it is possible to define security and audit policies, which enforce and log all access to data.

Oracle Database Manages All Types of Data

Information Lifecycle Management is concerned with all data in an organization. This includes not just structured data, such as orders in an OLTP system or a history of sales in a data warehouse, but also unstructured data, such as E-mail, documents, and images.

Although the Oracle Database already supports the storing of unstructured data through the use of BLOBs and Oracle Fast Files (available in Oracle Database 11*g*), a sophisticated document management system is available in Oracle Content Database, when used in conjunction with the Enterprise Edition. It includes role-based security to ensure that content is only accessed by authorized personnel and policies which describe what happens to the content during its lifetime.

Therefore, if all of the information in your organization is contained in an Oracle database, then you can take advantage of the features and functionality provided by the database to manage and move the data as it evolves during its lifetime, without having to manage multiple types of data stores.

Regulatory Requirements

Today, many organizations must retain specific data for a specific period of time. Failure to comply with these regulations could result in organizations having to pay very heavy fines. Therefore, around the world, a number of regulatory requirements, such as Sarbanes-Oxley, HIPAA, DOD5015.2-STD in the US and the European Data Privacy Directive in the European Union, are changing how organizations manage their data. These regulations specify what data must be retained, whether it can be changed, and for how long it must be retained, which could be for a period of 30 years or longer.

These regulations frequently demand that electronic data is secure from unauthorized access and changes, and that there is an audit trail of all changes to data and by whom. The Oracle Database can retain huge quantities of data without impacting application performance. It also contains the features required to restrict access and prevent unauthorized changes to data, and can be further enhanced with Oracle Database Vault and Oracle Audit Vault. The Oracle Database also provides cryptographic functions that can be used to demonstrate that a highly privileged user has not intentionally modified data. Flashback Data Archive can be used to show all the versions of a row during its lifetime.

Implementing ILM Using Oracle Database

Building an Information Lifecycle Management solution using the Oracle Database is quite straightforward and can be completed by following these four simple steps, although Step 4 is optional if ILM is not being implemented for compliance:

- Step 1: Define the Data Classes
- Step 2: Create Storage Tiers for the Data Classes
- Step 3: Create Data Access and Migration Policies
- Step 4: Define and Enforce Compliance Policies

Step 1: Define the Data Classes

In order to make effective use of Information Lifecycle Management, the first step is to look at all the data in your organization and determine:

- What data is important, where is it stored, and what needs to be retained
- How this data flows within the organization
- What happens to this data over time and whether it is still needed
- The degree of data availability and protection that is needed
- Data retention for legal and business requirements

Once there is an understanding of how the data is used, it can then be classified on this basis. The most common type of classification is by age or date, but other types are possible, such as by product or privacy. A hybrid classification could also be used, such as by privacy and age.

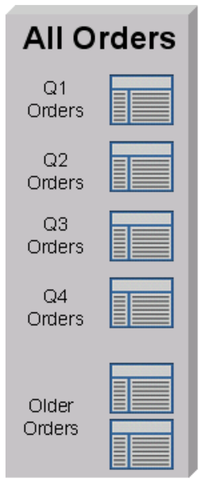
In order to treat the data classes differently, the data needs to be physically separated. When information is first created, it is often frequently accessed, but then over time it may be referenced very infrequently. For instance, when a customer places an order, they regularly look at the order to see its status and whether it has been shipped. Once it arrives, they may never reference that order again. This order would also be included in regular reports that are run to see what goods are being ordered, but, over time, would not figure in any of the reports and may only be referenced in the future if someone does a detailed analysis that involves this data. Therefore, orders could be classified by the Financial Quarters Q1, Q2, Q3, and Q4, and as Historical Orders.

The advantage of using this approach is that when the data is grouped at the row level by its class, which in this example would be the date of the order, all orders for Q1 can be managed as a self contained unit, where as the orders for Q2 would reside in a different class. This can be achieved by using partitioning. Since partitions are completely transparent to the application, the data is physically separated but the application still sees all of the orders.

Partitioning

Partitioning involves physically placing data according to a data value, and a frequently used technique is to partition information by date. Figure 5–1 illustrates a scenario where the orders for Q1, Q2, Q3, and Q4 are stored in individual partitions and the orders for previous years are stored in other partitions.

Figure 5–1 Allocating Data Classes to a Partition



Oracle offers several different partitioning methods. Range partitioning is one of the most frequently used partitioning methods for ILM. Interval and reference partitioning (introduced in Oracle Database 11*g*) are also particularly suited for use in an ILM environment.

There are a number of benefits to partitioning data. Partitioning provides an easy way to distribute the data across appropriate storage devices depending on its usage, while still keeping the data online and stored on the most cost-effective device. Since partitioning is completely transparent to anyone accessing the data, no application changes are required, thus partitioning can be implemented at any time. When new partitions are required, they are simply added using the `ADD PARTITION` clause or they can be created automatically if interval partitioning is being used.

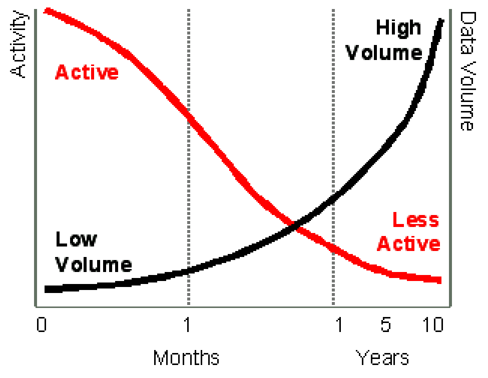
Among other benefits, each partition can have its own local index. When the optimizer uses partition pruning, queries will only access the relevant partitions instead of all partitions, thus improving query response times.

The Lifecycle of Data

An analysis of your data is likely to reveal that initially, it is accessed and updated on a very frequent basis. As the age of the data increases, its access frequency diminishes to almost negligible, if any. Therefore, most organizations find themselves in the situation where many users are accessing current data while very few users are accessing older data, as illustrated in Figure 5–2. Thus, data can be considered to be one of the following: active, less active, historical, or ready to be archived.

With so much data being held, during its lifetime the data should be moved to different physical locations. Depending on where the data is in its lifecycle, it needs to be located on the most appropriate storage device.

Figure 5–2 Data Usage Over Time



Step 2: Create Storage Tiers for the Data Classes

Since Oracle Database can take advantage of many different storage options, the next step is to establish the required storage tiers. Although you can create as many storage tiers as you require, a suggested starting point are the following tiers:

- High Performance

  The high performance storage tier is where all the important and frequently accessed data would be stored, such as the partition holding our Q1 orders. This would utilize smaller, faster disks on high performance storage devices.

- Low Cost

  The low cost storage tier is where the less frequently accessed data is stored, such as the partitions holding the orders for Q2, Q3, and Q4. This tier would be built using large capacity disks, such as those found in modular storage arrays or the low costs ATA disks, which offer the maximum amount of inexpensive storage.

- Online Archive

  The online archive storage tier is where all the data that is seldom accessed or modified would be stored. It is likely to be extremely large and to store the maximum quantity of data. Various techniques can be used to compress the data. This tier could be located in the database or it could be in another database, which serves as a central archive database for all information within the enterprise. Stored on low cost storage devices, such as ATA drives, the data would still be online and available, for a cost that is only slightly higher than storing this information on tape, without the disadvantages that come with archiving data to tape. If the Online Archive storage tier is identified as read-only, then it would be impossible to change the data and subsequent backups would not be required after the initial database backup.

- Offline Archive (optional)

  The offline archive storage tier is an optional tier because it is only used when there is a requirement to remove data from the database and store it in some other format, such as XML on a tape.

Figure 5–2 illustrates how data is used over a period of time. Using this information, it can be determined that to retain all this information, several storage tiers are required to hold all of the data, which also has the benefit of significantly reducing total storage costs.

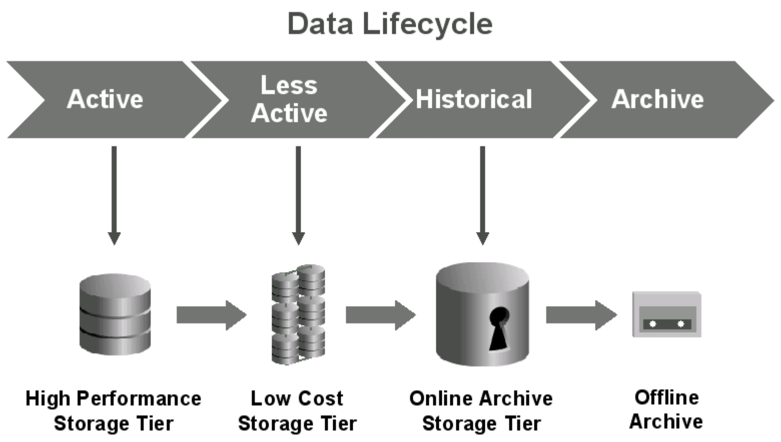
Once the storage tiers have been created, the data classes identified in "Step 1: Define the Data Classes" on page 5-3 will be physically implemented inside the database using partitions. This approach provides an easy way to distribute the data across the appropriate storage devices depending on its usage, while still keeping the data online and readily available, and stored on the most cost-effective device.

Note: Automatic Storage Management (ASM) can also be used to manage the data across the storage tiers.

Assigning Classes to Storage Tiers

Once the storage tiers have been defined, the data classes (partitions) identified in Step 1 can be assigned to the appropriate storage tiers. This provides an easy way to distribute the data across the appropriate storage devices depending on its usage, keeping the data online and available, and stored on the most cost-effective device. This is illustrated in Figure 5–3. Using this approach, no application changes are required because the data is still seen.

Figure 5–3 Data Lifecycle



The Costs Savings of using Tiered Storage

One of the benefits of implementing an ILM strategy is the cost savings that can result from using multiple tiered storage. Assume that we have 3 TB worth of data to store, comprised of: 200 GB on High Performance, 800 GB on Low Cost, and 2 TB on Online Archive. Assume the cost per GB is $72 on the High Performance tier, $14 on the Low Cost tier, and $7 on the Online Archive tier.

Table 5–1 illustrates the possible cost savings using tiered storage, rather than storing all data on one class of storage. As you can see, the cost savings can be quite significant and, if the data is suitable for database compression, then even further cost savings are possible.

Table 5–1 Cost Savings Using Tiered Storage

Storage Tier	Single Tier using High Performance Disks	Multiple Storage Tiers	Multiple Tiers with Database Compression
High Performance (200 GB)	$14,400	$14,400	$14,400
Low Cost (800 GB)	$57,600	$11,200	$11,200
Online Archive (2 TB)	$144,000	$14,000	$5,600
	$216,000	$39,600	$31,200

Step 3: Create Data Access and Migration Policies

The next step is to ensure that only authorized users have access to the data and to specify how to move the data during its lifetime. As the data ages, there are a number of techniques that can be used to migrate the data between the storage tiers.

Controlling Access to Data

The security of your data is another very important part of Information Lifecycle Management because the access rights to the data may change during its lifetime. In addition, there may be regulatory requirements that place exacting demands on how the data can be accessed.

The data in an Oracle Database can be secured using database features, such as:

- Database Security
- Views
- Virtual Private Database

Virtual Private Database (VPD) defines a very fine-grained level of access to the database. Security policies determine which rows may be viewed and the columns that are visible. Multiple policies can be defined so that different users and applications see different views of the same data. For example, the majority of users could see the information for Q1, Q2, Q3, and Q4, while only authorized users would be able to view the historical data.

A security policy is defined at the database level and is transparently applied to all database users. The benefit of this approach is that it provides a secure and controlled environment for accessing the data, which cannot be overridden and can be implemented without requiring any application changes. In addition, read-only tablespaces can be defined which ensures that the data will not change.

Moving Data using Partitioning

During its lifetime, data will need to be moved. This may occur for the following reasons:

- For performance, only a limited number of orders are held on high performance disks
- Data is no longer frequently accessed and is using valuable high performance storage, and needs to be moved to a low-cost storage device
- Legal requirements demand that the information is always available for a given period of time, and it needs to be held safely for the lowest possible cost

There are a number of ways that data can be physically moved in the Oracle Database to take advantage of the different storage tiers. For example, if the data is partitioned, then a partition containing the orders for Q2 could be moved online from the high performance storage tier to the low cost storage tier. Since the data is being moved within the database, it can be physically moved, without affecting the applications that require it or causing disruption to regular users.

Sometimes individual data items, rather than a group of data, must be moved. For example, suppose data was classified according to a level of privacy and a report, which was once secret, is now to be made available to the public. If the classification changed from secret to public and the data was partitioned on its privacy classification, then the row would automatically move to the partition containing public data.

Whenever data is moved from its original source, then it is very important to ensure that the process selected adheres to any regulatory requirements, such as, the data cannot be altered, is secure from unauthorized access, easily readable, and stored in an approved location.

Step 4: Define and Enforce Compliance Policies

The last step in an Information Lifecycle Management solution is the creation of policies for compliance. When data is decentralized and fragmented, compliance policies have to be defined and enforced in every data location, which could easily result in a compliance policy being overlooked. However, using the Oracle Database to provide a central location for storing data means that it is very easy to enforce compliance policies as they are all managed and enforced from one central location.

When defining compliance policies, consider the following areas:

- Data Retention
- Immutability
- Privacy
- Auditing
- Expiration

Data Retention

The retention policy describes how the data is to be retained, how long it must be kept, and what happens at the end of life. An example of a retention policy is a record must be stored in its original form, no modifications are allowed, it must be kept for seven years, and then it may be deleted. Using Oracle Database security, it is possible to ensure that data remains unchanged and that only authorized processes can remove the data at the appropriate time. Retention policies can also be defined via a lifecycle definition in the ILM Assistant.

Immutability

Immutability is concerned with proving to an external party that data is complete and has not been modified. Cryptographic or digital signatures can be generated by the Oracle Database and retained either inside or outside of the database, to show that data has not been altered.

Privacy

The Oracle Database provides several ways to ensure data privacy. Access to data can be strictly controlled through the use of security policies defined using Virtual Private Database (VPD). In addition, individual columns can be encrypted so that anyone looking at the raw data cannot see its contents.

Auditing

The Oracle Database has the ability to track all access and changes to data. These auditing capabilities can be defined either at the table level or through fine-grained auditing, which specifies the criteria for when an audit record is generated. Auditing can be further enhanced using Audit Vault.

Expiration

Ultimately, data may expire for business or regulatory reasons and would need to be removed from the database. The Oracle Database can remove data very quickly and efficiently by simply dropping the partition which contains the information identified for removal.

The Benefits of an Online Archive

There usually comes a point during the lifecycle of the data when it is no longer being regularly accessed and is considered eligible for archiving. Traditionally, at this time, the data would have been removed from the database and stored on tape, because it is capable of storing vast quantities of information for a very low cost. Today it is no longer necessary to archive that data to tape, instead it can remain in the database, or transferred to a central online archive database. All this information would be stored using low-cost storage devices whose cost per gigabyte is very close to that of tape.

There are a number of benefits to keeping all of the data in a database for archival purposes. The most important benefit is that the data will always be instantly available. Therefore, time is not wasted locating the tapes where the data was archived and determining whether the tape is readable and still in a format that can be loaded into the database.

If the data has been archived for many years, then development time may also be needed to write a program to reload the data into the database from the tape archive. This could prove to be expensive and time consuming, especially if the data is extremely old. If the data is retained in the database, then this is not a problem, because it is already online, and in the latest database format.

Holding the historical data in the database no longer impacts the time required to backup the database and the size of the backup. When RMAN is used to back up the database, it will only include in the backup the data that has changed. Since historical data is less likely to change, once the data has been backed up, it will not be backed up again.

Another important factor to consider is how the data is to be physically removed from the database, especially if it is to be transferred from a production system to a central database archive. Oracle provides the capability to move this data rapidly between databases by using transportable tablespaces or partitions, which moves the data as a complete unit.

When it is time to remove data from the database, the fastest way is to remove a set of data. This is achieved by keeping the data in its own partition. The partition can be dropped, which is a very fast operation. However, if this approach is not possible because data relationships must be maintained, then a conventional SQL delete statement must be issued. You should not underestimate the time required to issue the delete statement.

If there is a requirement to remove data from the database and there is a possibility that the data may need to be returned to the database in the future, then consider removing the data in a database format such as a transportable tablespace, or use the XML capability of the Oracle Database to extract the information in an open format.

Consider an online archive of your data into an Oracle database for the following reasons:

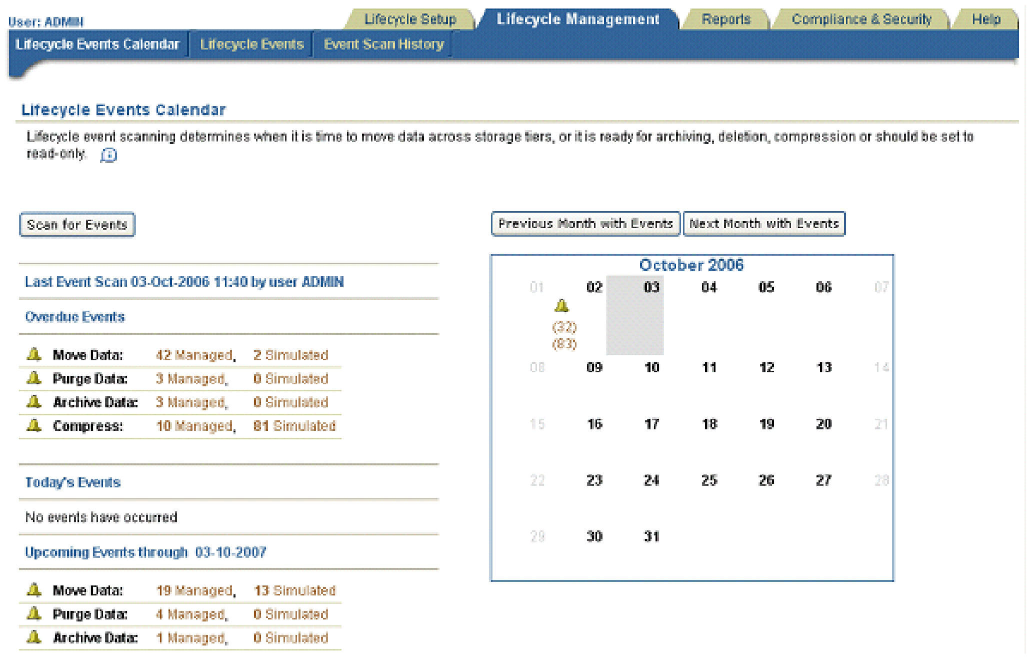
- The cost of disk is approaching that of tape, eliminate the time to find the tape that contains the data and the cost of restoring that data
- Data remains online when needed, faster access to meet business requirements
- Data online means immediate access, so cannot be fined by regulatory body for failing to produce data

- Use the current application to access the data, no need to waste resources to build a new application

Oracle ILM Assistant

The Oracle ILM Assistant provides a graphical user interface (GUI) for managing your ILM environment. Figure 5–4 shows the first screen of the ILM Assistant, which lists the outstanding tasks that should be performed.

Figure 5–4 ILM Assistant Initial Screen



Lifecycle Events Calendar

Lifecycle event scanning determines when it is time to move data across storage tiers, or it is ready for archiving, deletion, compression or should be set to read-only.

Last Event Scan 03-Oct-2006 11:40 by user ADMIN

Overdue Events

Move Data:	42 Managed,	2 Simulated
Purge Data:	3 Managed,	0 Simulated
Archive Data:	3 Managed,	0 Simulated
Compress:	10 Managed,	81 Simulated

Today's Events

No events have occurred

Upcoming Events through 03-10-2007

Move Data:	19 Managed,	13 Simulated
Purge Data:	4 Managed,	0 Simulated
Archive Data:	1 Managed,	0 Simulated

October 2006

01 (32) (83)	02	03	04	05	06	07
08	09	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

The ILM Assistant provides the ability to create lifecycle definitions, which are assigned to tables in the database. Using this lifecycle definition, the ILM Assistant advises when it is time to move, archive, or delete data, as shown by the calendar. It will also illustrate the storage requirements and cost savings associated with moving the data.

The ILM Assistant can manage only partitioned tables. For non-partitioned tables, the ILM Assistant generates a script to show how the table could be partitioned, and it also provides the capability to simulate partitioning on a table to view the actions that would arise if the table were partitioned.

The ILM Assistant will not execute any commands for the tasks it recommends to be performed, such as migrating data to different storage tiers. Instead, it generates a script of the commands that need to be executed.

To assist with managing compliance issues, the ILM Assistant shows all Virtual Private Databases (VPD) and Fine-Grained Audit (FGA) policies that have been defined on tables under ILM control. In addition, both Database and FGA audit records can be viewed and digital signatures generated and compared.

The Oracle ILM Assistant requires that Oracle Application Express is installed in the database where the tables to be managed by the ILM Assistant reside.

The ILM Assistant provides capability in the following areas:

- Lifecycle Setup

- Lifecycle Management
- Compliance & Security
- Reports

Lifecycle Setup

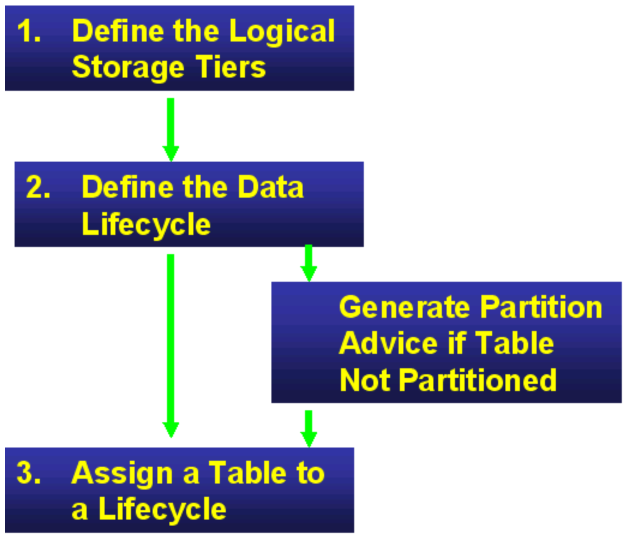
The **Lifecycle Setup** area of the ILM Assistant is comprised of the following tasks that need to be performed to prepare for managing your data:

- Logical Storage Tiers
- Lifecycle Definitions
- Lifecycle Tables
- Preferences

If this is the first time that you have used the ILM Assistant, then it is here where you specify exactly how the data is to be managed by the ILM Assistant. The following steps must be completed before the ILM Assistant is able to give advice on data placement, as illustrated in Figure 5–5.

1. Define the logical storage tiers
2. Define the lifecycle definitions
3. Select tables to be managed by the lifecycle definitions

Figure 5–5 ILM Assistant: Specifying How Data is Managed


1. Define the Logical Storage Tiers
2. Define the Data Lifecycle
Generate Partition Advice if Table Not Partitioned
3. Assign a Table to a Lifecycle

Other options available within setup include the ability to:

- View partition simulation
- View a lifecycle summary of mapped tables and their logical storage tiers and lifecycle definitions
- View storage costs
- Define policy notes
- Customize the ILM Assistant via preferences

Logical Storage Tiers

A logical storage tier is a name given to a logical group of storage devices; typically all disks of the same type will be identified by that name. For example, the group called High Performance could refer to all the high performance disks. Any number of logical storage tiers may be defined and the devices are identified by the assigned tablespaces, which reside upon them.

The Cost per GB value must be a value greater than zero. The value is used by the ILM Assistant to project storage costs when data is mapped to the tier. It is recommended that you enter a value that represents a reasonably accurate cost of storing data on the tier. This would include the physical purchase price of a device. However, you might also want to consider other associated costs, such as maintenance and running costs.

Each storage tier will have a set of assigned tablespaces that are labeled as a read-write preferred tablespace, read-only preferred tablespace, or a secondary tablespace. If read-write data can be migrated onto the tier, then the read-write preferred tablespace is required. If the storage tier will accept read-only data, then a read-only preferred tablespace must also be identified.

In addition to the preferred tablespaces, one or more secondary tablespaces may be assigned to the tier. Secondary tablespaces are typically located in the same location as the read-write preferred tablespace for the storage tier.

Since the ILM Assistant only supports a single preferred tablespace, any read-write data that must reside on the tier would generate a migration event to move the data to the read-write preferred tablespace. To avoid unnecessary data migration events, the ILM Assistant allows existing data to remain on a secondary tablespace for the storage tier.

Lifecycle Definitions

A lifecycle definition describes how data migrates across the logical storage tiers during its lifetime. It is comprised of one or more lifecycle stages that select a logical storage tier, data attributes such as compression and read only, and a duration for data residing on that lifecycle stage.

A lifecycle definition is valid if it contains at least one lifecycle stage. There must be a final stage, which is either user specified or automatically generated by the ILM Assistant upon completion of the lifecycle definition process. For the final stage you must specify what happens to data at lifecycle end.

A lifecycle definition is comprised of a number of stages that describes what happens to data during its lifetime. Lifecycle stages are initially created in reverse time order (that is, working backwards in time from the current date). Every stage must have a unique name; an optional description can be supplied.

If the stage is not the final stage, then you must specify how long the data is to remain on this stage and any stage attributes such as whether the data should be compressed or set to read only. Note that it is only possible to specify a read only stage if a preferred read only tablespace has been defined for the logical storage tier for this stage.

The current stage represents the present time but can span any length of time. A lifecycle can only have one current stage. The final stage is required as it describes what happens when data reaches its end-of-life. A lifecycle can only have one final stage and it is automatically created if the user does not create one. Possible actions are:

- Purge the data

- Archive the data off-line
- Allow the data to remain on-line

Stages that store data on-line also permit several attributes to be defined that affect the data. The supported attributes are:

- Compress
- Compress and Read-Only
- Read-Only

Each stage is comprised of the following information:

- Stage Type

  A stage is classified as a current stage, final stage, or unclassified.

- Stage Name

  Displays the user-supplied name of the stage.

- Stage Description

  Displays the user-supplied stage description.

- Action

  Displays the action performed when data maps to the stage. Possible actions are:

  – Remain Online
  – Archive Offline
  – Purge

- Tier Name

  Displays the storage tier associated with the stage. For a stage that purges data or moves data offline, a tier is not specified.

- Attributes

  Displays the optional data attributes that will be applied to data when it maps to the stage. Possible values are:

  – Compress
  – Compress and Read-Only
  – Read-Only

- Stage Duration

  Displays the length of time the data can remain mapped to the stage.

- Stage Start Date

  Displays the actual calendar date for the beginning of the stage. The date is computed based on the adjacent stages and the user-specified fiscal start date.

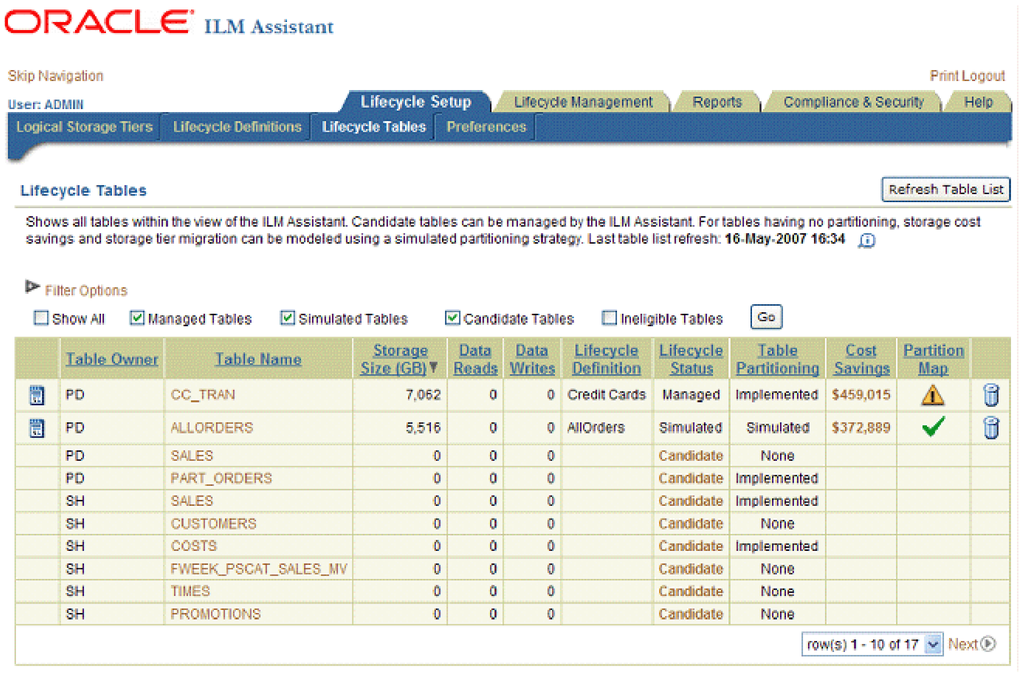
- Stage End Date

  Displays the actual calendar date for the end of the stage. The date is computed based on the adjacent stages and the user-specified fiscal start date.

Lifecycle Tables

The **Lifecycle Tables** area identifies those tables that may be managed by the ILM Assistant, and it is here where these tables are mapped to a lifecycle definition, as illustrated in Figure 5–6. A database may contain many tables, only some of which you wish to consider as candidates for ILM. A table is automatically eligible if it is range partitioned on a date column. When the table is associated with a lifecycle definition, the ILM Assistant can manage its data. For tables having no partitioning, storage cost savings and storage tier migration can be modeled using a simulated partitioning strategy.

Figure 5–6 ILM Assistant: Lifecycle Tables



Lifecycle Tables

Shows all tables within the view of the ILM Assistant. Candidate tables can be managed by the ILM Assistant. For tables having no partitioning, storage cost savings and storage tier migration can be modeled using a simulated partitioning strategy. Last table list refresh: **16-May-2007 16:34**

☐ Show All ☑ Managed Tables ☑ Simulated Tables ☑ Candidate Tables ☐ Ineligible Tables

	Table Owner	Table Name	Storage Size (GB) ▼	Data Reads	Data Writes	Lifecycle Definition	Lifecycle Status	Table Partitioning	Cost Savings	Partition Map	
	PD	CC_TRAN	7,062	0	0	Credit Cards	Managed	Implemented	$459,015	⚠	
	PD	ALLORDERS	5,516	0	0	AllOrders	Simulated	Simulated	$372,889	✓	
	PD	SALES	0	0	0		Candidate	None			
	PD	PART_ORDERS	0	0	0		Candidate	Implemented			
	SH	SALES	0	0	0		Candidate	Implemented			
	SH	CUSTOMERS	0	0	0		Candidate	None			
	SH	COSTS	0	0	0		Candidate	Implemented			
	SH	FWEEK_PSCAT_SALES_MV	0	0	0		Candidate	None			
	SH	TIMES	0	0	0		Candidate	None			
	SH	PROMOTIONS	0	0	0		Candidate	None			

row(s) 1 - 10 of 17 Next

If the table is not yet partitioned, then you will be directed to a **Partition Simulation** page where you can setup a full simulation. Similar to setting up a managed table, a simulation can be previewed and accepted on this page. Upon returning from the simulation page, the table is now eligible for full lifecycle management in simulation mode.

The difference between a managed table and a simulated table is that a managed table contains actual partitions while a simulated table only contains fake partitioning data. All reports and event detection work with both types of lifecycle tables. However, any table upon which partitioning is being simulated will only be seen as being partitioned from within the ILM Assistant. All other tools will continue to see it as a non-partitioned table.

Though the lifecycle tables view shows all accessible tables, the ILM Assistant may not be able to manage every table. In those cases, the table will be marked as ineligible and a link will be provided to explain the exception. Some examples of ineligible tables are:

- Tables having no date column
- Tables partitioned on non-date columns
- Tables partitioned using a partition type other than range
- Tables containing a LONG column
- Index-organized tables

The display for Lifecycle Tables can be customized to show managed, simulated, candidate, and ineligible tables, and is comprised of the following information:

- Table Owner

  The Oracle schema that owns the table
- Table Name

  The table that may allow ILM management
- Storage Size

  The current estimated size of the table. The value is scaled according to the Size Metric as specified within the Filter Options.
- Data Reads

  The current sum of all logical and physical reads for the table.
- Data Writes

  The current sum of all physical writes for the table.
- Lifecycle Definition

  If the ILM Assistant is managing the table, then the required lifecycle definition is displayed here.
- Lifecycle Status

  Provides the current status of the table. This will indicate whether the table is eligible, is managed, or is simulated. For tables that are ineligible, the status link provides an explanation regarding its incompatibility with the ILM Assistant.
- Table Partitioning

  Provides a status of the table partitioning. A table can have partitioning implemented, simulated, or none.
- Cost Savings

  When the ILM Assistant is managing a table, a total cost-savings value is computed and displayed here.
- Partition Map

  Indicates that the current table partitioning scheme is compatible with the lifecycle definition. Clicking on the icon displays a detailed report of the table partitions.

Lifecycle Table List For installations having many tables, the ILM Assistant provides a table list caching system to prevent long page waits and possible browser timeouts. The table list is a snapshot of all user tables on the system that should be periodically refreshed to maintain consistency within the ILM Assistant. Typically, the table list should be refreshed when application tables have been added, changed, or removed outside of the ILM Assistant, or when up-to-date table statistics are desired.

By default, a table list refresh operation will attempt to scan for every table defined in the database. For large application environments, this can take a long time to complete. Typically, ILM Assistant management of tables is limited to a small number of tables. To avoid refreshing the table list with the entire set of tables found in the database, filtering may be used to narrow the number of tables to be scanned. For example, if the user was only interested in managing tables in the `SH` schema, the Table Owner Filter can be set to `SH`. To estimate the time it may take to do refresh, click **Estimate Refresh Statistics**. This will return the projected number of tables that match the filters as well as the time it will take to process the data.

Purging unused entries in the cache will clean up a cache that contains any entries that are not currently managed by the ILM Assistant. It will not affect any of the tables that currently match the filters.

As a guideline, the ILM Assistant can refresh the table list at a rate of 300 to 350 tables per minute. The operation may be interrupted from the **Lifecycle Tables** screen. An interrupt will stop the refresh operation as if it has reached the normal end of the table scan. Because of the nature of the process, an interrupt can take up to 30 seconds to stop the actual scan operation.

Partition Map The **Partition Map** column in the Lifecycle Tables Report indicates whether all the partitions in the table will fit inside a stage and do not overlap stages. The Mapping Status indicates the quality of the partition-to-stage relationship. A green checkmark indicates the partition resides completely within the stage without violating date boundaries. A warning icon indicates some type of mismatch. Possible exceptions for the stage mapping are:

- Misaligned partitions

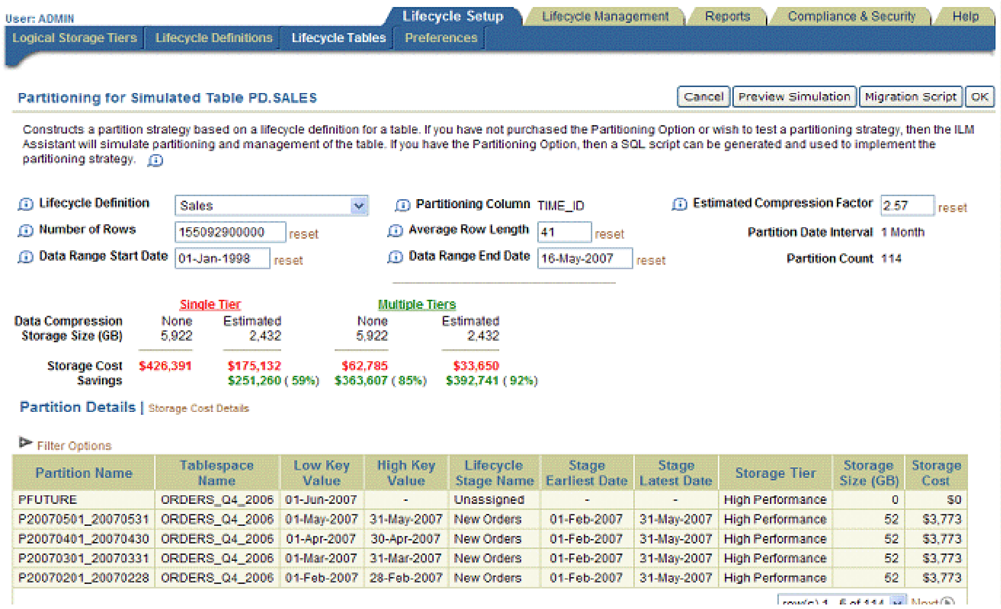
  A partition can be misaligned when it cannot fit into an entire stage. This can happen if the lifecycle stage duration is smaller than the smallest partition range. To resolve this, either choose a better lifecycle definition to manage the table or adjust the stage duration by editing the lifecycle definition.

- Tablespace is not associated with a logical storage tier

  This is very common for new ILM Assistant users. In order to perform cost analysis, the ILM Assistant needs to associate all referenced tablespaces with a tier. Typically, the easiest correction is to edit a logical storage tier and add the missing tablespace as a secondary tablespace.

Storage Costs The ILM Assistant provides a comprehensive storage cost and savings report associated with the managed or simulated table, as illustrated in Figure 5–7.

Figure 5–7 ILM Assistant: Partitioning for Simulated Tables



Partitioning for Simulated Table PD.SALES

Constructs a partition strategy based on a lifecycle definition for a table. If you have not purchased the Partitioning Option or wish to test a partitioning strategy, then the ILM Assistant will simulate partitioning and management of the table. If you have the Partitioning Option, then a SQL script can be generated and used to implement the partitioning strategy.

Lifecycle Definition	Sales	Partitioning Column	TIME_ID		Estimated Compression Factor	2.57	reset
Number of Rows	155092900000	reset	Average Row Length	41	reset	Partition Date Interval	1 Month
Data Range Start Date	01-Jan-1998	reset	Data Range End Date	16-May-2007	reset	Partition Count	114

	Single Tier		Multiple Tiers	
Data Compression	None	Estimated	None	Estimated
Storage Size (GB)	5,922	2,432	5,922	2,432
Storage Cost	$426,391	$175,132	$62,785	$33,650
Savings		$251,260 (59%)	$363,607 (85%)	$392,741 (92%)

Partition Name	Tablespace Name	Low Key Value	High Key Value	Lifecycle Stage Name	Stage Earliest Date	Stage Latest Date	Storage Tier	Storage Size (GB)	Storage Cost
PFUTURE	ORDERS_Q4_2006	01-Jun-2007	-	Unassigned	-	-	High Performance	0	$0
P20070501_20070531	ORDERS_Q4_2006	01-May-2007	31-May-2007	New Orders	01-Feb-2007	31-May-2007	High Performance	52	$3,773
P20070401_20070430	ORDERS_Q4_2006	01-Apr-2007	30-Apr-2007	New Orders	01-Feb-2007	31-May-2007	High Performance	52	$3,773
P20070301_20070331	ORDERS_Q4_2006	01-Mar-2007	31-Mar-2007	New Orders	01-Feb-2007	31-May-2007	High Performance	52	$3,773
P20070201_20070228	ORDERS_Q4_2006	01-Feb-2007	28-Feb-2007	New Orders	01-Feb-2007	31-May-2007	High Performance	52	$3,773

The report is divided into two main areas. The top portion of the report is a rollup showing the totals for the managed or simulated tables. For managed tables, there are two subsections that show data for a non-ILM environment using a single storage tier and an ILM managed, multi-tier environment. For simulated tables, a third section is provided that shows an ILM managed, multi-tier environment that includes the estimated effects of compression.

The bottom section of the storage costs page is the detail section that breaks up the cost areas by logical storage tier:

- Single-Tier Size

  Displays the total size of the entities. For a lifecycle-based report, the value represents the sum of all table sizes that are assigned the current lifecycle definition. For managed tables, the size is the actual size as indicated by the database storage statistics. For simulated tables, the size is the projected size as calculated by the user-specified number of rows and average row length.

- Single-Tier Cost

  Displays the single-tier cost, which is calculated by multiplying the single-tier size of the current entities by the cost of storing the data on the most expensive tier within the lifecycle definition.

- Cost per GB

  Displays the user-specified cost when setting up the storage tier. The value is used to calculate the storage costs for partitions that are assigned to the tier.

- Multi-Tier Size

  Displays the total size of the entities that reside on that tier. For lifecycles, it represents all table partitions that are associated with the current tier. For a table, it represents the sum of all partitions that are associated with the tier. The size does not include any projected compression.

- Multi-Tier Cost

  Displays the cost, which is calculated by multiplying the cost per gigabyte for the current tier by the space occupied by the entities. For lifecycles, it represents all table partitions that are associated with the current tier. For a table, it represents the sum of all partitions that are associated with the tier.

- Multi-Tier Savings

  Displays the savings, which is computed by subtracting the multi-tier cost from the calculated cost of storing the same data using the single-tier approach.

- Percent Savings

  Displays the ratio of multi-tier savings to the single-tier cost for the same data.

- Multi-Tier Compressed Size

  Displays the total size of the entities that reside on that tier. For lifecycles, it represents all table partitions that are associated with the current tier. For a table, it represents the sum of all partitions that are associated with the tier. The size includes projected compression based on the estimated compression factor assigned by the user.

  This report item is only present when viewing simulated table data.

- Multi-Tier Compressed Cost

  Displays the cost, which is calculated by multiplying the cost per gigabyte for the current tier by the space occupied by the entities. For lifecycles, it represents all table partitions that are associated with the current tier. For a table, it represents the sum of all partitions that are associated with the tier. The size includes projected compression based on the estimated compression factor assigned by the user.

This report item is only present when viewing simulated table data.

- Multi-Tier Compressed Savings

  Displays the savings, which is computed by subtracting the multi-tier compressed cost from the calculated cost of storing the same data using the single-tier approach.

  This report item is only present when viewing simulated table data.

- Percent Savings

  Displays the ratio of multi-tier compressed savings to the single-tier cost for the same data.

  This report item is only present when viewing simulated table data.

- Lifecycle Stages Compressed

  When setting up lifecycle stages, the user has the option of requiring the partitions to be compressed when assigned to the stage. This value shows the number of stages assigned to the storage tier that have the compressed attribute set.

- Partitions Compression

  Displays the number of partitions on the storage tier that are currently compressed.

Partition Simulation Implementing Partitioning is likely to be a major task for any organization and the ILM Assistant allows you to model the impact before actually reorganizing the data. To achieve this, the ILM Assistant requires the following information in simulation mode:

- Lifecycle Definition

  Select a lifecycle definition that will be used to manage the simulated table. The simulated partitions will be derived from the lifecycle stages defined in the lifecycle. The ILM Assistant will determine the optimal date range based on the stage duration information supplied.

- Partitioning Column

  Select a suitable date column as the partitioning key. If the current table has only one date column, then the column will automatically be selected and displayed in read-only form.

- Partition Date Interval

  Displays the optimal partition range interval based on the selected lifecycle definition. The ILM Assistant will compute an interval that will guarantee that all generated partitions will properly align with the lifecycle stages.

- Number of Rows

  Provide the number of rows in the current table. The default value is retrieved from the current tables database statistics. If the default value is unavailable, or you wish to project future growth, you may enter any value greater than zero.

- Average Row Length

  Provide the average row length for the table. The default value is retrieved from the current tables database statistics. If the statistics are not valid, then the ILM

Assistant will query the table and calculate a maximum row size. If the default value is unsuitable, or you wish to project future growth, then you may enter any value greater than zero.

- Estimated Compression Factor

  Provide a compression factor. The compression factor is used exclusively by the ILM Assistant to estimate storage costs and savings. The factor is purely an estimate, but can give you savings potential. A value of one indicates no compression is projected. A value greater than one indicates a reduction in space using the formula reduction = 1 / factor. The default value is calculated by sampling a small percentage of the table for compression potential.

An additional option after previewing the simulation is Migration Script generation, as illustrated in Figure 5–7 on page 5-16. This allows the user to create a script that can be used to convert the existing non-partitioned table to a partitioned counterpart. It should be noted that the script contains a simple create operation and a command to migrate the existing data; however, parts of the script have been commented out to prevent accidental operation. A conversion of a table to a partitioned table should be carefully planned.

Preferences

Preferences control various aspects of the ILM Assistant's behavior and display of data (for example, the default date format for most entered values and reports, or the default number of rows to display). The following preferences can be set:

- Compression sample block count
- Compression sample percent
- Date format (Long form)
- Date format (Short form)
- Demonstration Mode

  Specifies a factor that amplifies the actual table sizes. A value of one effectively disables the mode since multiplying a number by one does not change the original value.

- Language preference
- Lifecycle table view filter

  Specifies the default selection to view when visiting the Lifecycle Tables page. Values can be combined to indicate multiple types of tables. For example, 3 indicates that both managed and simulated tables are to be shown. Possible values are:

  1 - Managed Tables
  2 - Simulated Tables
  4 - Candidate Tables
  8 - Ineligible Tables

  The default value is 7, which excludes ineligible tables.

- Maximum report rows to display
- Maximum viewable tables
- Refresh rate for progress monitoring
- Report column maximum display length

- Start page for lifecycle setup

  Possible values are:

  – Logical Storage Tiers

  – Lifecycle Definitions

  – Lifecycle Tables

- Storage size metric

  Specifies the default size metric to be used when viewing storage size values. Possible values are:

  KB - Kilobytes
  MB - Megabytes
  GB - Gigabytes
  TB - Terabytes

  The value is case sensitive.

Lifecycle Management

Lifecycle Management is concerned with the tasks that must be performed to move data to the correct place in the Information Lifecycle. Information is available on the following:

- Lifecycle Events Calendar
- Lifecycle Events
- Event Scan History

Lifecycle Events Calendar

The **Lifecycle Events Calendar** shows the calendar of previous, current, and (optionally,) future lifecycle events that must be performed to place data at the appropriate place in the information lifecycle, as illustrated in Figure 5–5 on page 5-11. You can use the **Previous Month with Events** button to navigate to previous months containing lifecycle events.

To identify which data must be moved, click on the **Scan for Events** button which will ask whether to scan for all events up to the current day, or into the future. Additionally, you may choose to evaluate all tables or selected tables. The ILM Assistant will then compare the current location of data with where it should be stored in accordance with the lifecycle definition and recommend the appropriate movement. It will also advise if data should be compressed or set to read only as defined by the lifecycle definition. All the recommendations made by the ILM Assistant are applied to partitions only.

Lifecycle Events

The **Lifecycle Events** report shows details about data migration events and provides a way to generate scripts to perform their actions. You can select some or all of the displayed events by clicking the checkboxes in the first column. You need to select events to generate scripts or to dismiss events. To generate a script on the selected events, click the **Generate Script** button. To dismiss the selected events to make them permanently disappear, click the **Dismiss Selected Events** button.

The event summary shows the following pieces of information:

- Recommended Action

Indicates the type of event that was detected by the scan operation. Possible event types are:

– MOVE PARTITION

  Indicates that a partition should be moved from its current logical storage tier to a new logical storage tier. The movement is achieved by moving the partition from one tablespace to another.

– COMPRESSION

  Indicates that the partition should have data compression enabled.

– READ-ONLY

  Indicates that the partition should be set to read-only.

– PURGE

  Indicates that the partition should be physically deleted.

- Partition Name

  Describes the affected partition.

- Current Tier

  Describes the current location of the partition.

- Recommended Tier

  Describes the target storage tier for move operations.

- Cost Savings

  Indicates the potential storage cost savings if the event action were to be implemented.

- Table Owner and Name

  Describes the partition table owner and name.

- Event Date

  Indicates the date on which the action should be performed. For events that should have been resolved in the past, a single keyword **Past** is shown; fore events in the future, a calendar date is displayed.

- Event Details

  Provides a link to event details. This area describes lifecycle details that affected the scan operation.

When a partition requires several logical operations such as move and compress, the ILM Assistant displays the operations as separate events. However, in the script, the operations may be combined into a single SQL DDL statement.

The ILM Assistant currently does not have any archive capability. Therefore, selecting archive events generates a script that identifies which partitions should now be archived and lists them as comments.

Event Scan History

Any authorized user can invoke event scanning via the Lifecycle Events Calendar. Over time, tracking the scan activity can be quite difficult, so a history is made available.

The history report shows the following pieces of information:

- Scan Date
- Submitted by User
- Lowest Event Date
- Highest Event Date
- Table Owner and Name
- Number of Events
- Lifecycle Status

Compliance & Security

The **Compliance & Security** area shows everything that can be used to enforce security and help maintain compliance with the numerous regulations from around the world. It provides an area to:

- View Current Status
- Prove Immutability
- View Privacy & Security Policies
- View Auditing
- Manage Policy Notes

Current Status

Current status summarizes the status of all the various **Compliance & Security** features that are available. For example, it advises how many Virtual Private Database (VPD) policies have been defined, when a digital signature was last generated, and when a comparison of digital signatures was last performed.

Digital Signatures and Immutability

Some regulations stipulate that it must be shown that data has not changed since it was entered into the database. One of the techniques that can be used to prove that data has not been altered is to generate a digital signature.

Oracle Database provides the capability to generate a digital signature for a SQL result set. This can be generated inside the ILM Assistant and is achieved by creating a named SQL result set which includes the query to describe the collection of records. The digital signature is generated and is initially saved in a text file.

To show that the data records in a query have not been altered, a digital signature can be presented for a previously defined SQL query, and re-generated on today's data and the signatures compared, to show that the data has not changed since the digital signature was originally generated.

Privacy & Security

The **Privacy & Security** area enables you to view:

- A summary of privacy and security definitions for each ILM table
- Virtual Private Database (VPD) policies
- Security views on tables managed by the ILM Assistant
- Reports on the access privileges granted to users for tables managed by the ILM Assistant

By default, the Lifecycle Table Summary is shown and VPD policies and user access information are available by selecting the appropriate links.

Lifecycle Table Summary The **Lifecycle Table Summary** provides an overview for each table as to which features are being used in terms of VPD policies and table grants issued.

Virtual Private Database (VPD) Policies Using standard database privileges, it is possible to limit access to a table to certain users. However,such access allows users to read all information in that table. VPD Policies provide a finer level of control on who can access information. Using a VPD Policy, it is possible to write sophisticated functions, which define exactly which data is visible to a user.

For example, a policy could say that certain users can only view the last 12 months of data, while other users can view all of the data. Another policy could say that the only data visible is in the state where the office is located. Therefore, VPD Policies are an extremely powerful tool in controlling access to information. Only VPD policies that have been defined on tables that are being managed by the ILM Assistant are shown on the **VPD Policies** report.

Table Access by User The **Table Access by User** report provides a list of all the access privileges granted to users for tables that have been assigned to Lifecycle Definitions.

Auditing

Some regulations require that an audit trail be maintained of all access and changes to data. In the Oracle Database, several types of auditing are available: database and fine-grained. They each create their own audit records, which can be viewed as one consolidated report in the ILM Assistant that can be filtered on several criteria.

Within the auditing area on the ILM Assistant, it is possible to:

- View the Current Audit Status
- Manage Fine-Grained Audit Policies
- View Audit Records

Fine-Grained Auditing Policies Standard Auditing within the Oracle Database logs all types of access to a table. However, there may be instances when it is desirable to only audit an event when a certain condition is met (for example, the value of the transaction being altered is greater than $10,000). This type of auditing is possible using Fine-Grained Audit policies where an audit condition can be specified and an optional function can be called for more sophisticated processing.

View Auditing Records It is possible within the ILM Assistant to view both database and fine-grained audit records for tables mapped to Lifecycle Definitions in the ILM Assistant. An icon represents the type of audit record: database (indicated by a disc) or FGA. Use the Filter condition to filter the audit records that are displayed and click on the report heading to sort the data on that column.

By default, the ILM Assistant only displays audit records for the current day. To see audit records for previous days, you must use the filter options to specify a date range of records to display.

Policy Notes Policy notes provide textual documentation of your data management policies or anything that you wish to document with respect to managing data during its lifetime. Policy notes are informational only; they do not affect the tasks performed by the ILM Assistant. They can be used as a central place to describe your policies, as reminders, and as a way to prove that your policies are documented. They can also be used to document SLA (Service Level Agreements) and to document the compliance rules that you are trying to enforce.

Reports

The ILM Assistant offers a variety of reports on all aspects of managing the ILM environment, which include the following:

- Multi-Tier Storage Costs by Lifecycle or Table
- Logical Storage Tier Summary
- Partitions by Table or Storage Tier
- Lifecycle Retention Summary
- Data Protection Summary

Implementing an ILM System Manually

The following example illustrates how to manually create storage tiers and partition a table across those storage tiers and then setup a VPD policy on that database to restrict access to the online archive tier data.

```
REM Setup the tablespaces for the data

REM These tablespaces would be placed on a High Performance Tier
CREATE SMALLFILE TABLESPACE q1_orders DATAFILE 'q1_orders'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE q2_orders DATAFILE 'q2_orders'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE q3_orders DATAFILE 'q3_orders'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE q4_orders DATAFILE 'q4_orders'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on a Low Cost Tier
CREATE SMALLFILE TABLESPACE "2006_ORDERS" DATAFILE '2006_orders'
SIZE 5M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE "2005_ORDERS"  DATAFILE '2005_orders'
SIZE 5M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on the Online Archive Tier
CREATE SMALLFILE TABLESPACE "2004_ORDERS" DATAFILE '2004_orders'
SIZE 5M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE old_orders DATAFILE 'old_orders'
SIZE 15M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;
```

```
REM Now create the Partitioned Table
CREATE TABLE allorders (
    prod_id       NUMBER       NOT NULL,
    cust_id       NUMBER       NOT NULL,
    time_id       DATE         NOT NULL,
    channel_id    NUMBER       NOT NULL,
    promo_id      NUMBER       NOT NULL,
    quantity_sold NUMBER(10,2) NOT NULL,
    amount_sold   NUMBER(10,2) NOT NULL)
 --
 -- table wide physical specs
 --
  PCTFREE 5 NOLOGGING
 --
 -- partitions
 --
 PARTITION BY RANGE (time_id)
  ( partition allorders_pre_2004 VALUES LESS THAN
     (TO_DATE('2004-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE old_orders,
    partition allorders_2004 VALUES LESS THAN
     (TO_DATE('2005-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE "2004_ORDERS",
    partition allorders_2005 VALUES LESS THAN
     (TO_DATE('2006-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE "2005_ORDERS",
    partition allorders_2006 VALUES LESS THAN
     (TO_DATE('2007-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE "2006_ORDERS",
    partition allorders_q1_2007 VALUES LESS THAN
     (TO_DATE('2007-04-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q1_orders,
    partition allorders_q2_2007 VALUES LESS THAN
     (TO_DATE('2007-07-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q2_orders,
    partition allorders_q3_2007 VALUES LESS THAN
     (TO_DATE('2007-10-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q3_orders,
    partition allorders_q4_2007 VALUES LESS THAN
     (TO_DATE('2008-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q4_orders);

ALTER TABLE allorders ENABLE ROW MOVEMENT;
```

```
REM Here is another example using INTERVAL partitioning

REM These tablespaces would be placed on a High Performance Tier
CREATE SMALLFILE TABLESPACE cc_this_month DATAFILE 'cc_this_month'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE cc_prev_month DATAFILE 'cc_prev_month'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on a Low Cost Tier
CREATE SMALLFILE TABLESPACE cc_prev_12mth DATAFILE 'cc_prev_12'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on the Online Archive Tier
CREATE SMALLFILE TABLESPACE cc_old_tran DATAFILE 'cc_old_tran'
SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;



REM Credit Card Transactions where new partitions automatically are placed on the
high performance tier
CREATE TABLE cc_tran (
    cc_no       VARCHAR2(16) NOT NULL,
    tran_dt     DATE         NOT NULL,
    entry_dt    DATE         NOT NULL,
    ref_no      NUMBER       NOT NULL,
    description VARCHAR2(30) NOT NULL,
    tran_amt    NUMBER(10,2) NOT NULL)
 --
 -- table wide physical specs
 --
 PCTFREE 5 NOLOGGING
 --
 -- partitions
 --
 PARTITION BY RANGE (tran_dt)
 INTERVAL (NUMTOYMINTERVAL(1,'month') ) STORE IN (cc_this_month )
  ( partition very_old_cc_trans VALUES LESS THAN
     (TO_DATE('1999-07-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_old_tran ,
    partition old_cc_trans VALUES LESS THAN
     (TO_DATE('2006-07-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_old_tran ,
    partition last_12_mths VALUES LESS THAN
     (TO_DATE('2007-06-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_prev_12mth,
    partition recent_cc_trans VALUES LESS THAN
    (TO_DATE('2007-07-01 00:00:00'
            ,'SYYYY-MM-DD HH24:MI:SS'
            ,'NLS_CALENDAR=GREGORIAN'
            )) TABLESPACE cc_prev_month,
    partition new_cc_tran VALUES LESS THAN
     (TO_DATE('2007-08-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_this_month);


REM Create a Security Policy to allow user SH to see all credit card data,
REM PM only sees this years data,
REM and all other uses cannot see the credit card data

CREATE OR REPLACE FUNCTION ilm_seehist
  (oowner IN VARCHAR2, ojname IN VARCHAR2)
   RETURN VARCHAR2 AS con VARCHAR2 (200);
BEGIN
  IF SYS_CONTEXT('USERENV','CLIENT_INFO') = 'SH'
  THEN -- sees all data
    con:= '1=1';
  ELSIF SYS_CONTEXT('USERENV','CLIENT_INFO') = 'PM'
  THEN -- sees only data for 2007
    con := 'time_id > ''31-Dec-2006''';
  ELSE
    -- others nothing
    con:= '1=2';
  END IF;
  RETURN (con);
END ilm_seehist;
/

REM Then the policy is added with the DBMS_RLS package as follows:

BEGIN
  DBMS_RLS.ADD_POLICY ( object_schema=>'SYSTEM'
                      , object_name=>'cc_tran'
                      , policy_name=>'ilm_view_history_data'
                      , function_schema=>'SYSTEM'
                      , policy_function=>'ilm_seehist'
                      , sec_relevant_cols=>'tran_dt'
                      );
END;
/
```

6 Using Partitioning in a Data Warehouse Environment

Data warehouses often contain large tables and require techniques both for managing these large tables and for providing good query performance across these large tables. This chapter describes the partitioning features that significantly enhance data access and improve overall application performance. This is especially true for applications that access tables and indexes with millions of rows and many gigabytes of data.

This chapter contains the following topics:

- What Is a Data Warehouse?
- Scalability
- Performance
- Manageability

What Is a Data Warehouse?

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

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

See Also: *Oracle Database Data Warehousing Guide*

Scalability

Partitioning helps scaling a data warehouse by dividing database objects into smaller pieces, enabling access to smaller, more manageable objects. Having direct access to smaller objects addresses the scalability requirements of data warehouses:

- Bigger Databases
- Bigger Individual tables: More Rows in Tables
- More Users Querying the System
- More Complex Queries

Bigger Databases

The ability to split a large database object into smaller pieces transparently provides benefits to manage a larger total database size. You can identify and manipulate individual partitions and subpartitions in order to cope with large database objects. Consider the following advantages of partitioned objects:

- Backup and recovery can be performed on a low level of granularity to cope with the size of the database.
- Part of a database object can be stored compressed while other parts can remain uncompressed.
- Partitioning can be used to store data transparently on different storage tiers to lower the cost of storing vast amounts of data. Refer to Chapter 5, "Using Partitioning for Information Lifecycle Management".

Bigger Individual tables: More Rows in Tables

It takes longer to scan a big table than it takes to scan a small table. Queries against partitioned tables may access one or more partitions that are small compared with the total size of the table. Similarly, queries may take advantage of partition elimination on indexes. It takes less time to read a smaller portion of an index from disk than to read the entire index. Index structures that share the partitioning strategy with the table, local partitioned indexes, can be accessed and maintained on a partition-by-partition basis.

The database can take advantage of the distinct data sets in separate partitions if you use parallel execution to speed up queries, DML, and DDL statements. Individual parallel execution servers can work on their own data set, identified by the partition boundaries.

More Users Querying the System

With partitioning, users are more likely to hit isolated and smaller data sets. As a result, the database will be able to return results faster than if all users hit the same and much larger data sets. Data contention is less likely.

More Complex Queries

Smaller data sets help perform complex queries faster. If smaller data sets are being accessed, then complex calculations are more likely to be processed in memory which is beneficial from a performance perspective and which reduces the application's I/O requirements. A larger set may have to be written to the temporary tablespace in order to complete, in which case additional I/O against the database storage occurs.

Performance

Good performance is a key to success for a data warehouse. Analyses run against the database should return within a reasonable amount of time, even if the queries access large amounts of data in tables that are terabytes in size. Partitioning provides fundamental functionality to enable successful data warehouses that are not prohibitively expensive in terms of hardware cost.

Partition Pruning

Partition pruning is an essential performance feature for data warehouses. In partition pruning, the optimizer analyzes FROM and WHERE clauses in SQL statements to eliminate unneeded partitions when building the partition access list. This enables Oracle Database to perform operations only on those partitions that are relevant to the SQL statement.

Partition pruning dramatically reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance and optimizing resource utilization.

> **See Also:** Chapter 4, "Partitioning for Availability, Manageability, and Performance" for more information about partition pruning and the difference between static and dynamic partition pruning

Basic Partition Pruning Techniques

The optimizer uses a wide variety of predicates for pruning. The three predicate types, equality, range, and IN-list, are the most commonly used cases of partition pruning. As an example, consider the following query:

```
SELECT SUM(amount_sold) day_sales
FROM sales
WHERE time_id = TO_DATE('02-JAN-1998', 'DD-MON-YYYY');
```

Because there is an equality predicate on the partitioning column of sales, this query will prune down to a single predicate and this will be reflected in the explain plan, as shown:

```
-----------------------------------------------------------------------------------------------
|  Id | Operation                  | Name   | Rows| Bytes | Cost (%CPU)| Time      |Pstart| Pstop |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |        |     |       | 21 (100)   |           |      |       |
|   1 |  SORT AGGREGATE            |        | 1   | 13    |            |           |      |       |
|   2 |   PARTITION RANGE SINGLE   |        | 485 | 6305  | 21 (10)    | 00:00:01  | 5    | 5     |
| * 3 |    TABLE ACCESS FULL       | SALES  | 485 | 6305  | 21 (10)    | 00:00:01  | 5    | 5     |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  3 - filter("TIME_ID"=TO_DATE('1998-01-02 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Similarly, a range or an IN-list predicate on the time_id column and the optimizer would be used to prune to a set of partitions. The partitioning type plays a role in which predicates can be used. Range predicates cannot be used for pruning on hash partitioned tables while they can be used for all other partitioning strategies. However, on list-partitioned tables, range predicates may not map to a contiguous set of partitions. Equality and IN-list predicates can be used to prune with all the partitioning methods.

Advanced Partition Pruning Techniques

Oracle also prunes in the presence of more complex predicates or SQL statements involving partitioned tables. A common situation is when a partitioned table is joined to the subset of another table, limited by a WHERE condition. For example, consider the following query:

```
SELECT t.day_number_in_month, SUM(s.amount_sold)
FROM sales s, times t
WHERE s.time_id = t.time_id
```

```
  AND t.calendar_month_desc='2000-12'
GROUP BY t.day_number_in_month;
```

If the database performed a nested loop join with `times` on the right hand side, then the query would only access the partition corresponding to this row from the `times` table, so pruning would implicitly take place. But, if the database performed a hash or sort merge join, this would not be possible. If the table with the `WHERE` predicate is relatively small compared to the partitioned table, and the expected reduction of records or partitions for the partitioned table is significant, then the database will perform dynamic partition pruning using a recursive subquery. The decision whether or not to invoke subquery pruning is an internal cost-based decision of the optimizer.

A sample plan using a hash join operation would look like the following:

```
-----------------------------------------------------------------------------------------------
| Id| Operation                    |  Name |  Rows | Bytes| Cost (%CPU)|  Time   | Pstart | Pstop |
-----------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT             |       |       |      | 761 (100)  |         |        |       |
|  1|  HASH GROUP BY               |       |    20 | 640  | 761 (41)   |00:00:10|        |       |
|* 2|   HASH JOIN                  |       | 19153 | 598K | 749 (40)   |00:00:09|        |       |
|* 3|    TABLE ACCESS FULL         | TIMES |    30 |  570 |  17 (6)    |00:00:01|        |       |
|  4|     PARTITION RANGE SUBQUERY |       |  918K | 11M  |   655 (33) |00:00:08| KEY(SQ)|KEY(SQ)|
|  5|      TABLE ACCESS FULL       | SALES |   918 | 11M  |   655 (33) |00:00:08| KEY(SQ)|KEY(SQ)|
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------
  2 - access("S"."TIME_ID"="T"."TIME_ID")
  3 - filter("T"."CALENDAR_MONTH_DESC"='2000-12')
```

This plan shows that dynamic partition pruning occurred on the `sales` table using a subquery, as can be seen from the `KEY(SQ)` value in the `PSTART` and `PSTOP` columns.

Another example using advanced pruning is the following, which uses an `OR` predicate:

```
SELECT p.promo_name promo_name, (s.profit - p.promo_cost) profit
FROM
   promotions p,
   (SELECT
      promo_id,
      SUM(sales.QUANTITY_SOLD * (costs.UNIT_PRICE - costs.UNIT_COST)) profit
   FROM
      sales, costs
   WHERE
      ((sales.time_id BETWEEN TO_DATE('01-JAN-1998','DD-MON-YYYY',
                  'NLS_DATE_LANGUAGE = American') AND
      TO_DATE('01-JAN-1999','DD-MON-YYYY', 'NLS_DATE_LANGUAGE = American')
OR
      (sales.time_id BETWEEN TO_DATE('01-JAN-2001','DD-MON-YYYY',
                  'NLS_DATE_LANGUAGE = American') AND
      TO_DATE('01-JAN-2002','DD-MON-YYYY', 'NLS_DATE_LANGUAGE = American')))
      AND sales.time_id = costs.time_id
      AND sales.prod_id = costs.prod_id
   GROUP BY
      promo_id) s
WHERE s.promo_id = p.promo_id
ORDER BY profit
DESC;
```

This query joins the `sales` and `costs` tables in the `sh` sample schema. The `sales` table is partitioned by range on the column `time_id`. One of the conditions in the query are two predicates on `time_id`, which are combined with an `OR` operator. This `OR` predicate is used to prune the partitions in `sales` table and a single join between the `sales` and `costs` table is performed. The plan is as follows:

```
----------------------------------------------------------------------------------------------------
| Id| Operation                  |  Name     |Rows |Bytes |TmpSp|Cost(%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT           |           | 4   |  200 |     | 3556 (14)| 00:00:43|       |       |
|  1|  SORT ORDER BY             |           | 4   |  200 |     | 3556 (14)| 00:00:43|       |       |
|* 2|   HASH JOIN                |           | 4   |  200 |     | 3555 (14)| 00:00:43|       |       |
|  3|    TABLE ACCESS FULL       |PROMOTIONS| 503  | 16599|     |    16 (0)| 00:00:01|       |       |
|  4|    VIEW                    |           |   4 |   68 |     | 3538 (14)| 00:00:43|       |       |
|  5|     HASH GROUP BY          |           |   4 |  164 |     | 3538 (14)| 00:00:43|       |       |
|  6|      PARTITION RANGE OR    |           | 314K|   12M|     |  3321 (9)| 00:00:40|KEY(OR)|KEY(OR)|
|* 7|       HASH JOIN            |           | 314K|   12M| 440K|  3321 (9)| 00:00:40|       |       |
|* 8|        TABLE ACCESS FULL| SALES        | 402K| 7467K|     |  400 (39)| 00:00:05|KEY(OR)|KEY(OR)|
|  9| TABLE ACCESS FULL          | COSTS     |82112| 1764K|     |   77 (24)| 00:00:01|KEY(OR)|KEY(OR)|
----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  2 - access("S"."PROMO_ID"="P"."PROMO_ID")
  7 - access("SALES"."TIME_ID"="COSTS"."TIME_ID" AND "SALES"."PROD_ID"="COSTS"."PROD_ID")
  8 - filter("SALES"."TIME_ID"<=TO_DATE('1999-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
      "SALES"."TIME_ID">=TO_DATE('1998-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') OR
      "SALES"."TIME_ID">=TO_DATE('2001-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
      "SALES"."TIME_ID"<=TO_DATE('2002-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

The database also does additional pruning when a column is range partitioned on multiple columns. As long as the database can guarantee that a particular predicate cannot be satisfied in a particular partition, the partition will be skipped. This allows the database to optimize cases where there are range predicates on more than one column or in the case where there are no predicates on a prefix of the partitioning columns.

See Also: "Partition Pruning Tips" on page 4-7 for tips on partition pruning

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 the use of both CPU and memory resources.

Partition-wise joins can be full or partial. Oracle decides which type of join to use.

Full Partition-Wise Joins

Full partition-wise joins can occur if two tables that are co-partitioned on the same key are joined in a query. The tables can be co-partitioned at the partition level, or at the subpartition level, or at a combination of partition and subpartition levels. Reference partitioning is an easy way to guarantee co-partitioning. Full partition-wise joins can be executed in serial and in parallel.

See Also: Chapter 4, "Partitioning for Availability, Manageability, and Performance" for more information on partition-wise joins

The following example shows a full partition-wise join on `orders` and `order_items`, in which the `order_items` table is reference partitioned.

```
CREATE TABLE orders
( order_id     NUMBER(12) NOT NULL
, order_date   DATE NOT NULL
, order_mode   VARCHAR2(8)
, order_status VARCHAR2(1)
, CONSTRAINT orders_pk PRIMARY KEY (order_id)
)
PARTITION BY RANGE (order_date)
( PARTITION p_before_jan_2006 VALUES LESS THAN (TO_
DATE('01-JAN-2006','dd-MON-yyyy'))
, PARTITION p_2006_jan VALUES LESS THAN (TO_DATE('01-FEB-2006','dd-MON-yyyy'))
, PARTITION p_2006_feb VALUES LESS THAN (TO_DATE('01-MAR-2006','dd-MON-yyyy'))
, PARTITION p_2006_mar VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
, PARTITION p_2006_apr VALUES LESS THAN (TO_DATE('01-MAY-2006','dd-MON-yyyy'))
, PARTITION p_2006_may VALUES LESS THAN (TO_DATE('01-JUN-2006','dd-MON-yyyy'))
, PARTITION p_2006_jun VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
, PARTITION p_2006_jul VALUES LESS THAN (TO_DATE('01-AUG-2006','dd-MON-yyyy'))
, PARTITION p_2006_aug VALUES LESS THAN (TO_DATE('01-SEP-2006','dd-MON-yyyy'))
, PARTITION p_2006_sep VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
, PARTITION p_2006_oct VALUES LESS THAN (TO_DATE('01-NOV-2006','dd-MON-yyyy'))
, PARTITION p_2006_nov VALUES LESS THAN (TO_DATE('01-DEC-2006','dd-MON-yyyy'))
, PARTITION p_2006_dec VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
)
PARALLEL;

CREATE TABLE order_items
( order_id NUMBER(12) NOT NULL
, product_id NUMBER NOT NULL
, quantity NUMBER NOT NULL
, sales_amount NUMBER NOT NULL
, CONSTRAINT order_items_orders_fk FOREIGN KEY (order_id) REFERENCES
orders(order_id)
)
PARTITION BY REFERENCE (order_items_orders_fk)
PARALLEL;
```

A typical data warehouse query would scan a large amount of data. Note that in the underlying plan, the columns `Rows`, `Bytes`, `Cost (%CPU)`, `Time`, and `TQ` have been removed.

```
EXPLAIN PLAN FOR
SELECT o.order_date
, sum(oi.sales_amount) sum_sales
FROM orders o
, order_items oi
WHERE o.order_id = oi.order_id
AND o.order_date BETWEEN TO_DATE('01-FEB-2006','DD-MON-YYYY')
                     AND TO_DATE('31-MAY-2006','DD-MON-YYYY')
GROUP BY o.order_id
, o.order_date
ORDER BY o.order_date;

---------------------------------------------------------------------------------------------
| Id  | Operation                         | Name        | Pstart| Pstop |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |             |       |       |      |            |
|   1 |  PX COORDINATOR                   |             |       |       |      |            |
|   2 |   PX SEND QC (ORDER)              | :TQ10001    |       |       | P->S | QC (ORDER) |
|   3 |    SORT GROUP BY                |              |       |       | PCWP |            |
|   4 |     PX RECEIVE                  |              |       |       | PCWP |            |
|   5 |      PX SEND RANGE              | :TQ10000     |       |       | P->P | RANGE      |
|   6 |       SORT GROUP BY             |              |       |       | PCWP |            |
|   7 |        PX PARTITION RANGE ITERATOR|            |     3 |     6 | PCWC |            |
|*  8 |         HASH JOIN               |              |       |       | PCWP |            |
|*  9 |          TABLE ACCESS FULL      | ORDERS       |     3 |     6 | PCWP |            |
|  10 |          TABLE ACCESS FULL      | ORDER_ITEMS  |     3 |     6 | PCWP |            |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   8 - access("O"."ORDER_ID"="OI"."ORDER_ID")
   9 - filter("O"."ORDER_DATE"<=TO_DATE(' 2006-05-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Partial Partition-Wise Joins

Oracle Database can perform partial partition-wise joins only in parallel. Unlike full partition-wise joins, partial partition-wise joins require you to partition only one table on the join key, not both tables. The partitioned table is referred to as the reference table. The other table may or may not be partitioned. Partial partition-wise joins are more common than full partition-wise joins.

To execute a partial partition-wise join, the database dynamically repartitions the other table based on the partitioning of the reference table. Once the other table is repartitioned, the execution is similar to a full partition-wise join.

The following example shows a call detail records table, `cdrs`, in a typical data warehouse scenario. The table is interval-hash partitioned.

```
CREATE TABLE cdrs
( id                  NUMBER
, cust_id             NUMBER
, from_number         VARCHAR2(20)
, to_number           VARCHAR2(20)
, date_of_call        DATE
, distance            VARCHAR2(1)
, call_duration_in_s  NUMBER(4)
) PARTITION BY RANGE(date_of_call)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY HASH(cust_id)
SUBPARTITIONS 16
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2005','dd-MON-yyyy')))
PARALLEL;
```

The `cdrs` table is joined with the non-partitioned `callers` table on `cust_id` to rank the customers who spent most time calling.

```
EXPLAIN PLAN FOR
SELECT c.cust_id
,      c.cust_last_name
,      c.cust_first_name
,      AVG(call_duration_in_s)
,      COUNT(1)
,      DENSE_RANK() OVER
       (ORDER BY (AVG(call_duration_in_s) * COUNT(1)) DESC) ranking
FROM   callers c
,      cdrs    cdr
WHERE cdr.cust_id = c.cust_id
```

```
AND cdr.date_of_call BETWEEN TO_DATE('01-JAN-2006','dd-MON-yyyy')
                         AND TO_DATE('31-DEC-2006','dd-MON-yyyy')
GROUP BY c.cust_id
, c.cust_last_name
, c.cust_first_name
ORDER BY ranking;
```

The execution shows a partial partition-wise join. Note that the columns `Rows`, `Bytes`, `Cost (%CPU)`, `Time`, and `TQ` have been removed.

```
--------------------------------------------------------------------------------------------
| Id  | Operation                            | Name     | Pstart| Pstop |IN-OUT| PQ Distrib |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |          |       |       |      |            |
|   1 |  WINDOW NOSORT                       |          |       |       |      |            |
|   2 |   PX COORDINATOR                     |          |       |       |      |            |
|   3 |    PX SEND QC (ORDER)                | :TQ10002 |       |       | P->S | QC (ORDER) |
|   4 |     SORT ORDER BY                    |          |       |       | PCWP |            |
|   5 |      PX RECEIVE                      |          |       |       | PCWP |            |
|   6 |       PX SEND RANGE                  | :TQ10001 |       |       | P->P | RANGE      |
|   7 |        HASH GROUP BY                 |          |       |       | PCWP |            |
|*  8 |         HASH JOIN                    |          |       |       | PCWP |            |
|   9 |          PART JOIN FILTER CREATE     | :BF0000  |       |       | PCWP |            |
|  10 |           BUFFER SORT                |          |       |       | PCWC |            |
|  11 |            PX RECEIVE                |          |       |       | PCWP |            |
|  12 |             PX SEND PARTITION (KEY)  | :TQ10000 |       |       | S->P | PART (KEY) |
|  13 |              TABLE ACCESS FULL       | CALLERS  |       |       |      |            |
|  14 |          PX PARTITION RANGE ITERATOR|          |   367 |   731 | PCWC |            |
|  15 |           PX PARTITION HASH ALL      |          |     1 |    16 | PCWC |            |
|* 16 |            TABLE ACCESS FULL         | CDRS     |  5857 | 11696 | PCWP |            |
--------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   8 - access("CDR"."CUST_ID"="C"."CUST_ID")
  16 - filter("CDR"."DATE_OF_CALL">=TO_DATE(' 2006-01-01 00:00:00', 'syyyy-mm-dd
hh24:mi:ss') AND "CDR"."DATE_OF_CALL"<=TO_DATE('
              2006-12-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Benefits of Partition-Wise Joins

Partition-wise joins offer benefits described in the following sections:

- Reduction of Communications Overhead
- Reduction of Memory Requirements

Reduction of Communications Overhead When executed in parallel, partition-wise joins reduce communications overhead. This is because, in the default case, parallel execution of a join operation by a set of parallel execution servers requires the redistribution of each table on the join column into disjoint subsets of rows. These disjoint subsets of rows are then joined pair-wise by a single parallel execution server.

The database can avoid redistributing the partitions because the two tables are already partitioned on the join column. This enables each parallel execution server to join a pair of matching partitions. This improved performance from using parallel execution is even more noticeable in Oracle Real Application Clusters configurations with internode parallel execution.

Partition-wise joins dramatically reduce interconnect traffic. Using this feature is key for large DSS configurations that use Oracle Real Application Clusters. Currently, most Oracle Real Application Clusters platforms, such as MPP and SMP clusters, provide limited interconnect bandwidths compared with their processing powers. Ideally, interconnect bandwidth should be comparable to disk bandwidth, but this is seldom the case. As a result, most join operations in Oracle Real Application Clusters experience high interconnect latencies without parallel execution of partition-wise joins.

Reduction of Memory Requirements Partition-wise joins require less memory than the equivalent join operation of the complete data set of the tables being joined. In the case of serial joins, the join is performed at the same time on a pair of matching partitions. If data is evenly distributed across partitions, then the memory requirement is divided by the number of partitions. There is no skew.

In the parallel case, memory requirements depend on the number of partition pairs that are joined in parallel. For example, if the degree of parallelism is 20 and the number of partitions is 100, then 5 times less memory is required because only 20 joins of two partitions are performed at the same time. The fact that partition-wise joins require less memory has a direct effect on performance. For example, the join probably does not need to write blocks to disk during the build phase of a hash join.

Performance Considerations for Parallel Partition-Wise Joins

The optimizer weighs the advantages and disadvantages when deciding whether or not to use partition-wise joins.

- In range partitioning where partition sizes differ, data skew increases response time; some parallel execution servers take longer than others to finish their joins. Oracle recommends the use of hash partitioning and subpartitioning to enable partition-wise joins because hash partitioning, if the number of partitions is a power of two, limits the risk of skew. Ideally the hash partitioning key is unique or almost unique to minimize the risk of skew.
- The number of partitions used for partition-wise joins should, if possible, be a multiple of the number of query servers. With a degree of parallelism of 16, for example, you can have 16, 32, or even 64 partitions. If there is an odd number of partitions, then some parallel execution servers are used less than others. For example, if there are 17 evenly distributed partition pairs, only one pair will work on the last join, while the other pairs will have to wait. This is because, in the beginning of the execution, each parallel execution server works on a different partition pair. At the end of this first phase, only one pair is left. Thus, a single parallel execution server joins this remaining pair while all other parallel execution servers are idle.

Sometimes, parallel joins can cause remote I/Os. For example, on Oracle Real Application Clusters environments running on MPP configurations, if a pair of matching partitions is not collocated on the same node, a partition-wise join requires extra internode communication due to remote I/O. This is because Oracle must transfer at least one partition to the node where the join is performed. In this case, it is better to explicitly redistribute the data than to use a partition-wise join.

Indexes and Partitioned Indexes

Indexes are optional structures associated with tables that allow SQL statements to execute more quickly against a table. Even though table scans are very common in many data warehouses, indexes can often speed up queries. The most commonly used indexes in a data warehouse are B-tree and bitmap indexes.

Both B-tree and bitmap indexes can be created as LOCAL indexes on a partitioned table, in which case they inherit the table's partitioning strategy. B-tree indexes can be created as global partitioned indexes on partitioned and on non-partitioned tables.

See Also: Chapter 4, "Partitioning for Availability, Manageability, and Performance" for more information about partitioned indexes

Local Partitioned Indexes

In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition. A local index is equipartitioned with the underlying table. Oracle partitions the index on the same columns as the underlying table, creates the same number of partitions or subpartitions, and gives them the same partition bounds as corresponding partitions of the underlying table.

Oracle also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced. This ensures that the index remains equipartitioned with the table.

For data warehouse applications, local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key. The following example creates a local B-tree index on a partitioned `customers` table.

```
ALTER SESSION enable parallel ddl;

CREATE INDEX cust_last_name_ix
ON customers(last_name) LOCAL
PARALLEL NOLOGGING ;
```

Bitmap indexes use a very efficient storage mechanism for low cardinality columns. Bitmap indexes are commonly used in data warehouses, especially in data warehouses that implement so-called star schemas. A single star schema consists of a central large fact table and multiple smaller dimension tables that describe the data in the fact table.

For example, the `sales` table in the sample `sh` schema in the Oracle Database is a fact table, that is described by dimension tables `customers`, `products`, `promotions`, `times`, and `channels`. Bitmap indexes enable the so-called star transformation, an optimization for fast query retrieval against star or star look-a-like schemas.

Fact table foreign key columns are ideal candidates for bitmap indexes, because generally there are relatively few distinct values relative to the total number of rows. Fact tables are often range or range-* partitioned, in which case you have to create local bitmap indexes. Global bitmap indexes on partitioned tables are not supported.

The following example creates a local partitioned bitmap index on the `sales` table.

```
ALTER SESSION enable parallel ddl;

CREATE BITMAP INDEX prod_id_ix
ON sales(prod_id) LOCAL
PARALLEL NOLOGGING;
```

See Also: *Oracle Database Data Warehousing Guide* for more information about the star transformation

Non-Partitioned Indexes

You can create non-partitioned indexes on non-partitioned tables and on partitioned tables. Non-partitioned indexes are primarily used on non-partitioned tables in data warehouse environments. You can use a non-partitioned global index on a partitioned table to enforce a primary or unique key. A non-partitioned (global) index can be useful for queries that commonly retrieve very few rows based on equality predicates or in-list on a column or set of columns that is not included in the partitioning key. In those cases, it can be faster to scan a single index than to scan many index partitions to find all matching rows.

Unique indexes on columns other than the partitioning columns must be global because unique local nonprefixed indexes whose key does not contain the partitioning key are not supported. Unique keys are not always enforced in data warehouses due to the controlled data load processes and the performance cost of enforcing the unique constraint. Global indexes can grow very large on tables with billions of rows.

The following example creates a global unique index on the `sales` table. Note that very few queries will benefit from this index. In systems with a very limited data load window, you should consider not to create and maintain it.

```
ALTER SESSION enable parallel ddl;

CREATE UNIQUE INDEX sales_unique_ix
ON sales(cust_id, prod_id, promo_id, channel_id, time_id)
PARALLEL NOLOGGING;
```

Note: Most partition maintenance operations invalidate non-partitioned indexes, forcing an index rebuild.

Global Partitioned Indexes

You can create global partitioned indexes on non-partitioned tables and on partitioned tables. In a global partitioned index, the keys in a particular index partition may refer to rows stored in more than one underlying table partition or subpartition. A global index can be range or hash partitioned, though it can be defined on any type of partitioned table.

A global index is created by specifying the `GLOBAL` attribute. The database administrator is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be merged or split as necessary.

Global indexes can be useful if there is a class of queries that uses an access path to the table to retrieve a few rows via an index, and by partitioning the index you can eliminate large portions of the index for the majority of queries that use the index. On a partitioned table you would consider a global partitioned index if the column or columns you should include to achieve partition pruning do not include the table partitioning key.

The following example creates a global hash-partitioned index on the `sales` table.

```
CREATE INDEX cust_id_prod_id_global_ix
ON sales(cust_id,prod_id)
GLOBAL PARTITION BY HASH (cust_id)
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
)
PARALLEL NOLOGGING;
```

Note: Most partition maintenance operations invalidate global partitioned indexes, forcing an index rebuild.

Partitioning and Data Compression

Data in a partitioned table can be compressed on a partition-by-partition basis. Using compressed data is most efficient for data that does not change frequently. Although Oracle Database 11*g* supports compression for all DML operations, it is still more efficient to modify data in a non-compressed table.

Common data warehouse scenarios often see few data changes as data ages and other scenarios only insert data. Using the partition management features, you can compress data on a partition-by-partition basis. Note that altering a partition to enable compression only applies to future data to be inserted into the partition. If you want to compress the existing data in the partition, then you have to move the partition. Enabling compression and moving a partition can be done in a single operation.

If you want to use table compression on partitioned tables with bitmap indexes, then you need to do the following before you introduce the compression attribute for the first time:

1. Mark bitmap indexes unusable.
2. Set the compression attribute.
3. Rebuild the indexes.

The first time you make a compressed partition part of an already existing, fully uncompressed partitioned table, you must either drop all existing bitmap indexes or mark them `UNUSABLE` prior to adding a compressed partition. This must be done regardless of whether any partition contains data. It is also independent of the operation that causes one or more compressed partitions to become part of the table. This does not apply to a partitioned table having B-tree indexes only.

The following example shows how to compress the `SALES_1995` partition in the `sales` table.

```
ALTER SESSION enable parallel ddl;

ALTER TABLE sales
MOVE PARTITION sales_1995
COMPRESS FOR ALL OPERATIONS
PARALLEL NOLOGGING;
```

If a table or a partition takes less space on disk, then the performance of large table scans in an I/O-constraint environment may improve.

Materialized Views and Partitioning

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 in a data warehouse are meant to speed up query performance.

The database supports transparent rewrites against materialized views, so that you do not need to modify the original queries to take advantage of precalculated results in materialized views. Instead of executing the query, the database will retrieve precalculated results from one or more materialized views, perform any necessary additional operations on the data, and return the query results. The database guarantees correct results in line with your setting of the `QUERY_REWRITE_INTEGRITY` initialization parameter.

See Also: *Oracle Database Data Warehousing Guide*

Partitioned Materialized Views The underlying storage for a materialized view is a table structure. You can partition materialized views like you can partition tables. When the database rewrites a query to run against materialized views, the query can take advantage of the same performance features that queries running against tables directly benefit from. The rewritten query may eliminate materialized view partitions. If joins back to tables or with other materialized views are necessary to retrieve the query result, then the rewritten query can take advantage of partition-wise joins.

The following example shows how to create a compressed partitioned materialized view that aggregates sales results to country level. This materialized view benefits from queries that summarize sales numbers by country level or higher to subregion or region level.

```
ALTER SESSION ENABLE PARALLEL DDL;

CREATE MATERIALIZED VIEW country_sales
PARTITION BY HASH (country_id)
PARTITIONS 16
COMPRESS FOR ALL OPERATIONS
PARALLEL NOLOGGING
ENABLE QUERY REWRITE
AS SELECT co.country_id
, co.country_name
, co.country_subregion
, co.country_region
, sum(sa.quantity_sold) country_quantity_sold
, sum(sa.amount_sold) country_amount_sold
FROM sales sa
, customers cu
, countries co
WHERE sa.cust_id = cu.cust_id
AND cu.country_id = co.country_id
GROUP BY co.country_id
, co.country_name
, co.country_subregion
, co.country_region;
```

See Also: *Oracle Database Data Warehousing Guide*

Manageability

Data Warehouses store historical data. An important part of a data warehouse is the data load and purge. Partitioning is powerful technology that can help data management for data warehousing.

Partition Exchange Load

Partitions can be added using Partition Exchange Load (PEL). When you use PEL you create a separate table that looks exactly like a single partition, including the same indexes and constraints, if any. If you use a composite partitioned table, then your separate table must use a partitioning strategy that matches the subpartitioning strategy of your composite partitioned table. You can then swap out an existing table partition with this separate table. In a data load scenario, data can be loaded into the separate table. Build indexes and implement constraints on the separate table, without impacting the table users query. Then perform the PEL, which is a very low-impact transaction compared to the data load. Daily loads, in conjunction with a range partition strategy by day, are common in data warehouse environments.

The following example shows a partition exchange load for the `sales` table.

```
ALTER TABLE sales ADD PARTITION p_sales_jun_2007
VALUES LESS THAN (TO_DATE('01-FEB-2007','dd-MON-yyyy'));

CREATE TABLE sales_jun_2007 COMPRESS FOR ALL OPERATIONS
AS SELECT * FROM sales WHERE 1=0;
```

Next, populate table `sales_jun_2007` with sales numbers for June 2007, and create the equivalent bitmap indexes and constraints that have been implemented on the `sales` table.

```
CREATE BITMAP INDEX time_id_jun_2007_bix ON sales_jun_2007(time_id)
NOLOGGING;
CREATE BITMAP INDEX cust_id_jun_2007_bix ON sales_jun_2007(cust_id)
NOLOGGING;
CREATE BITMAP INDEX prod_id_jun_2007_bix ON sales_jun_2007(prod_id)
NOLOGGING;
CREATE BITMAP INDEX promo_id_jun_2007_bix ON sales_jun_2007(promo_id)
NOLOGGING;
CREATE BITMAP INDEX channel_id_jun_2007_bix ON sales_jun_2007(channel_id)
NOLOGGING;

ALTER TABLE sales_jun_2007 ADD CONSTRAINT prod_id_fk FOREIGN KEY (prod_id)
REFERENCES products(prod_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT cust_id_fk FOREIGN KEY (cust_id)
REFERENCES customers(cust_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT promo_id_fk FOREIGN KEY (promo_id)
REFERENCES promotions(promo_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT time_id_fk FOREIGN KEY (time_id)
REFERENCES times(time_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT channel_id_fk FOREIGN KEY
(channel_id) REFERENCES channels(channel_id);
```

Next, exchange the partition.

```
ALTER TABLE sales
EXCHANGE PARTITION p_sales_jun_2007
WITH TABLE sales_jun_2007
INCLUDING INDEXES;
```

See Also: Chapter 3, "Partition Administration" for more information about partition exchange load

Partitioning and Indexes

Local indexes are easiest when performing partition maintenance operations. Local indexes do not invalidate a global index when partition management takes place. Use `INCLUDING INDEXES` in the PEL statement in order to exchange the local indexes with the equivalent indexes on the separate table so that no index partitions get invalidated. In the case of PEL, you can update global indexes as part of the load. Use the `UPDATE GLOBAL INDEXES` extension to the PEL command. If an index requires updating, then the PEL takes much longer.

Partitioning and Materialized View Refresh Strategies

There are different ways to keep materialized views updated:

- Full refresh
- Fast (incremental) refresh based on materialized view logs against the base tables
- Manually using DML, followed by `ALTER MATERIALIZED VIEW CONSIDER FRESH`

In order to enable query rewrites, set the `QUERY_REWRITE_INTEGRITY` initialization parameter. If you manually keep materialized views up to date, then you must set `QUERY_REWRITE_INTEGRITY` to either `TRUSTED` or `STALE_TOLERATED`.

See Also: *Oracle Database Data Warehousing Guide*

If your materialized views and base tables use comparable partitioning strategies, then PEL can be an extremely powerful way to keep materialized views up-to-date manually. For example, if both your base table and your materialized view use range partitioning, then you can consider PEL to keep your base table and materialized view up-to-date. The total data refresh scenario would work as follows:

- Create tables to enable PEL against the tables and materialized views
- Load data into the tables, build the indexes, and implement any constraints
- Update the base tables using PEL
- Update the materialized views using PEL
- Execute `ALTER MATERIALIZED VIEW CONSIDER FRESH` for every materialized view you updated using this strategy

Note that this strategy implies a short period of time, in between PEL against the base table and PEL against the materialized view, in which the materialized view does not reflect the current data in the underlying tables. Take into account the `QUERY_REWRITE_INTEGRITY` setting and the activity on your system to identify whether you can cope with this situation.

See Also: *Oracle Database 2 Day + Data Warehousing Guide* for an example of this refresh scenario

Removing Data from Tables

Data Warehouses commonly keep a time window of data. For example, 3 years worth of historical data is stored.

Partitioning makes it very easy to purge data from a table. You can use the `DROP PARTITION` or `TRUNCATE PARTITION` statements in order to purge data. Common strategies also include using a partition exchange load to unload the data from the table and replacing the partition with an empty table before dropping the partition. Archive the separate table you exchanged before emptying or dropping it.

Note that a drop or truncate operation would invalidate a global index or a global partitioned index. Local indexes remain valid. The local index partition is dropped when you drop the table partition.

The following example shows how to drop partition `sales_1995` from the `sales` table.

```
ALTER TABLE sales
DROP PARTITION sales_1995
UPDATE GLOBAL INDEXES PARALLEL;
```

Partitioning and Data Compression

Data in a partitioned table can be compressed on a partition-by-partition basis. Using compressed data is most efficient for data that does not change frequently. Common data warehouse scenarios often see few data changes as data ages and other scenarios only insert data. Using the partition management features, you can compress data on a partition-by-partition basis.

If a table takes less space on disk, then performance of large table scans in an I/O-constraint environment may improve.

Gathering Statistics on Large Partitioned Tables

In order to get good SQL execution plans, it is important to have reliable table statistics. Oracle automatically gathers statistics using the statistics job that is activated upon database installation, or you can manually gather statistics using the `DBMS_STATS` package. Managing statistics on large tables is more challenging than managing statistics on smaller tables.

If a query accesses only a single table partition, then it is best to have partition-level statistics. If queries perform some partition elimination, but not down to a single partition, then you should gather both partition-level statistics and global statistics. Oracle Database 11*g* can maintain global statistics for a partitioned table incrementally. Only partitions that have changed are scanned and not the entire table.

A typical scenario for statistics management on a partitioned table is the use of Partition Exchange Load (PEL). If you add data using PEL and you do not plan to update the global-level statistics as part of the data load, then you should gather statistics on the table the data was initially loaded into, before you exchange it with the partition. Your global-level statistics will become stale after the partition exchange. When you re-gather the global-level statistics, or when the automatic statistics gather job regathers the global-level statistics, only the new partition, and not the entire table, will be scanned.

7

Using Partitioning in an Online Transaction Processing Environment

Partitioning was initially adopted to cope with the performance requirements for data warehouses. With the explosive growth of OLTP systems and their user populations, partitioning is particularly useful for OLTP systems as well.

Partitioning is often used for OLTP systems to reduce contention in order to support a very large user population. It also helps in addressing regulatory requirements facing OLTP systems, including storing larger amounts of data in a cost-effective manner. This chapter contains the following topics:

What is an OLTP System?

Online Transaction Processing (OLTP) systems are one of the most common data processing systems in today's enterprises. Classical examples of OLTP systems are order entry, retail sales, and financial transaction systems.

OLTP systems are primarily characterized through a specific data usage that is different from data warehousing environments, yet some of the characteristics, such as having large volumes of data and lifecycle-related data usage and importance, are identical.

The main characteristics of an OLTP environment are:

- Short response time

  The nature of OLTP environments is predominantly any kind of interactive ad hoc usage, such as telemarketeers entering telephone survey results. OLTP systems require short response times in order for users to remain productive.

- Small transactions

  OLTP systems normally read and manipulate highly selective, small amounts of data; the data processing is mostly simple and complex joins are relatively rare. There is always a mix of queries and DML workload. For example, one of many call center employees retrieves customer details for every call and enters customer complaints while reviewing past communication with the customer.

- Data maintenance operations

It is not uncommon to have reporting programs and data updating programs that need to run either periodically or on an ad hoc basis. These programs, which run in the background while users continue to work on other tasks, may require a large number of data-intensive computations. For example, a University may start batch jobs assigning students to classes while students can still sign up online for classes themselves.

- Large user populations

  OLTP systems can have immeasurably large user populations where many users are trying to access the same data at once. For example, an online auction Web site can have hundreds of thousands (if not millions) of users accessing data on its Web site at the same time.

- High concurrency

  Due to the large user population, the short response times, and small transactions, the concurrency in OLTP environments is very high. A requirement for thousands of concurrent users is not uncommon.

- Large data volumes

  Depending on the application type, the user population, and the data retention time, OLTP systems can become very large. For example, every customer of a bank could have access to the online banking system which shows all their transactions for the last 12 months.

- High availability

  The availability requirements for OLTP systems are often extremely high. An unavailable OLTP system can impact a very large user population, and organizations can suffer major losses if OLTP systems are unavailable. For example, a stock exchange system has extremely high availability requirements during trading hours.

- Lifecycle related data usage

  Similar to data warehousing environments, OLTP systems often experience different data access patterns over time. For example, at the end of the month, monthly interest is calculated for every active account.

The following are benefits of partitioning for OLTP environments:

- Support for bigger databases

  Backup and recovery, as part of a high availability strategy, can be performed on a low level of granularity to cope with the size of the database. OLTP systems usually remain online during backups and users may continue to access the system while the backup is running. The backup process should not introduce major performance degradation for the online users.

  Partitioning helps to reduce the space requirements for the OLTP system because part of a database object can be stored compressed while other parts can remain uncompressed. Update transactions against uncompressed rows are more efficient than updates on compressed data.

  Partitioning can be used to store data transparently on different storage tiers to lower the cost of storing vast amounts of data.

- Partition maintenance operations for data maintenance (instead of DML)

  In the case of data maintenance operations (purging being the most common operation), you can leverage partition maintenance operations in conjunction with

Oracle's capability of online index maintenance. A partition management operation generates less redo than the equivalent DML operations.

- Potential higher concurrency through elimination of hot spots

  A common scenario for OLTP environments is to have monotonically increasing index values that are used to enforce primary key constraints, thus creating areas of high concurrency and potential contention: every new insert tries to update the same set of index blocks. Partitioned indexes, in particular hash-partitioned indexes, can help to alleviate this situation.

Performance

Performance in OLTP environments heavily relies on performant index access, thus the choice of the most appropriate index strategy becomes crucial. The following section discusses best practices for deciding whether or not to partition indexes in an OLTP environment.

Deciding Whether or not to Partition Indexes

Due to the selectivity of queries and high concurrency of OLTP applications, the choice of the right index strategy is indisputably one of the most important decisions for the use of partitioning in an OLTP environment. The following basic rules help to understand the main benefits and trade-offs for the various possible index structures:

- A non-partitioned index, while larger than individual partitioned index segments, always leads to a single index probe (or scan) if an index access path is chosen; there is only one segment for a table. The data access time and number of blocks being accessed is identical for both a partitioned and a non-partitioned table.

  A non-partitioned index does not provide partition autonomy and requires an index maintenance operation for every partition maintenance operation that affects rowids (for example, drop, truncate, move, merge, coalesce, or split operations).

- With partitioned indexes, there are always multiple segments. Whenever the Oracle Database cannot prune down to a single index segment, the database has to touch more than one segment. This potentially leads to higher I/O requirements (*n* index segment probes compared with one probe with a nonpartitioned index) and can have an impact (measurable or not) on the runtime performance. This is true for all partitioned indexes.

  Partitioned indexes can either be local partitioned indexes or global partitioned indexes. Local partitioned indexes always inherit the partitioning key from the table and are fully aligned with the table partitions. Consequently, any kind of partition maintenance operation requires little to no index maintenance work. For example, dropping or truncating a partition does not incur any measurable overhead for index maintenance; the local index partitions will be either dropped or truncated.

  Partitioned indexes that are not aligned with the table are called global partitioned indexes. Unlike local indexes, there is no relation between a table and an index partition. Global partitioned indexes give the flexibility to choose a partitioning key that is most optimal for a performant partition index access. Partition maintenance operations normally affect more (if not all) partitions of a global partitioned index, depending on the operation and partitioning key of the index.

- Under some circumstances, having multiple segments for an index can be beneficial for performance. It is very common in OLTP environments to leverage sequences to create artificial keys; consequently you create key values that are monotonically increasing, which results in many insert processes competing for the same index blocks. Introducing a global partitioned index (for example, using global hash partitioning on the key column) can alleviate this situation. If you have, for example, four hash partitions for such an index, then you now have four index segments you are inserting data into, reducing the concurrency on these segments by a factor of four for the insert processes.

With less contention, the application can support a larger user population. The following example shows the creation of a unique index on the `order_id` column of the `orders` table. The `order_id` in the OLTP application is filled using a sequence number. The unique index uses hash partitioning in order to reduce contention for the monotonically increasing `order_id` values. The unique key is then used to create the primary key constraint.

```
CREATE UNIQUE INDEX orders_pk
ON orders(order_id)
GLOBAL PARTITION BY HASH (order_id)
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
) NOLOGGING;

ALTER TABLE orders ADD CONSTRAINT orders_pk
PRIMARY KEY (order_id)
USING INDEX;
```

Enforcing uniqueness is important database functionality for OLTP environments. Uniqueness can be enforced with non-partitioned as well as with partitioned indexes. However, since partitioned indexes provide partition autonomy, the following requirements must be met to implement unique indexes:

- A non-partitioned index can enforce uniqueness for any given column or combination of columns. The behavior of a non-partitioned index is no different for a partitioned table compared to a non-partitioned table.
- Each partition of a partitioned index is considered an autonomous segment. To enforce the autonomy of these segments, you always have to include the partitioning key columns as a subset of the unique key definition.
  - Unique global partitioned indexes must always be prefixed with the partitioning columns.
  - Unique local indexes must have the partitioning key of the table as a subset of the unique key definition.

Using Index-Organized Tables

When your workload fits the use of index-organized tables, then you have to consider how to use partitioning on your index-organized table and on any secondary indexes.

See Also:

- *Oracle Database Administrator's Guide* for more information about index-organized tables
- Chapter 3, "Partition Administration" for more information on how to create partitioned index-organized tables

Whether or not to partition secondary indexes on index-organized tables follows the same considerations as indexes on regular heap tables. You can partition an index-organized table, but the partitioning key must be a subset of the primary key. A common reason to partition an index-organized table is to reduce contention; this is typically achieved using hash partitioning.

Another reason to partition an index-organized table is to be able to physically separate data sets based on one of the primary key columns. For example, an application hosting company can physically separate application instances for different customers by list partitioning on the company identifier. Queries in such a scenario can often take advantage of index partition pruning, shortening the time for the index scan. ILM scenarios with index-organized tables and partitioning are less common because they require a date column to be part of the primary key.

Manageability

In addition to the performance benefits, Partitioning also enables the optimal data management for large objects in an OLTP environment. Every partition maintenance operation in the Oracle Database can be extended to atomically include global and local index maintenance, enabling the execution of any partition maintenance operation without affecting the 24x7 availability of an OLTP environment.

Partition maintenance operations in OLTP systems occur often because of ILM scenarios. In these scenarios, [range | interval] partitioned tables, or [range | interval]-* composite partitioned tables, are common.

Other business cases for partition maintenance operations include scenarios surrounding the separation of application data. For example, a retail company runs the same application for multiple branches in a single schema. Depending on the branch revenues, the application (as separate partitions) is stored on more performant storage. List partitioning, or list-* composite partitioning, is a common partitioning strategy for this type of business case.

Hash partitioning, or hash subpartitioning for tables, can be used in OLTP systems to obtain similar performance benefits to the performance benefits achieved in data warehouse environments. The majority of the daily OLTP workload consists of relatively small operations, executed in serial. Periodic batch operations, however, may execute in parallel and benefit from the distribution benefits hash partitioning and subpartitioning can provide for partition-wise joins. For example, end-of-the-month interest calculation may be executed in parallel in order to complete within a nightly batch window.

See Also: Chapter 4, "Partitioning for Availability, Manageability, and Performance" for more information about the performance benefits of partitioning

Impact of a Partition Maintenance Operation on a Partitioned Table with Local Indexes

Whenever a partition maintenance operation takes place, Oracle locks the affected table partitions for any DML operation. Data in the affected partitions, with the exception of a `DROP` or `TRUNCATE` operation, is still fully accessible for any `SELECT` operation. Since local indexes are logically coupled with the table (data) partitions, only the local index partitions of the affected table partitions have to be maintained as part of a partition maintenance operation, enabling the most optimal processing for the index maintenance.

For example, when you move an older partition from a high end storage tier to a low cost storage tier, the data and the index are always available for `SELECT` operations;

the necessary index maintenance is either to update the existing index partition to reflect the new physical location of the data or, more commonly, a move and rebuild of the index partition to a low cost storage tier as well. If you drop an older partition after you have archived it, then its local index partitions get dropped as well, enabling a split-second partition maintenance operation that only affects the data dictionary.

Impact of a Partition Maintenance Operation on Global Indexes

Whenever a global index is defined on a partitioned or non-partitioned table, there is no correlation between a distinct table partition and the index. Consequently, any partition maintenance operation affects all global indexes or index partitions. As with tables containing local indexes, the affected partitions are locked to prevent DML operations against the affected table partitions. However, unlike the index maintenance for local indexes, any global index will still be fully available for DML operations and will not affect the online availability of the OLTP system. Conceptually and technically, the index maintenance for global indexes for a partition maintenance operation is comparable to the index maintenance that would become necessary for a semantically identical DML operation.

For example, dropping an old partition is semantically equivalent to deleting all the records of the old partition using the SQL `DELETE` statement. In both cases, all index entries of the deleted data set have to be removed from any global index as a normal index maintenance operation which will not affect the availability of an index for `SELECT` and DML operations. In this scenario, a drop operation represents the most optimal approach: data is removed without the overhead of a conventional `DELETE` operation and the global indexes are maintained in a non-intrusive manner.

Common Partition Maintenance Operations in OLTP Environments

The two most common partition maintenance operations are the removal of data and the relocation of data onto lower cost storage tier devices.

Removing (Purging) Old Data

Using either a `DROP` or `TRUNCATE` operation will remove older data based on the partitioning key criteria. The drop operation will remove the data as well as the partition metadata, while a truncate operation will only remove the data but preserve the metadata. All local index partitions are dropped respectively, as well as truncated. Normal index maintenance will be done for partitioned or non-partitioned global indexes and is fully available for select and DML operations.

The following example drops all data older than January 2006 from the `orders` table. Note that as part of the drop statement, an `UPDATE GLOBAL INDEXES` statement is executed, so that the global index remains usable throughout the maintenance operation. Any local index partitions are dropped as part of this operation.

```
ALTER TABLE orders DROP PARTITION p_before_jan_2006
UPDATE GLOBAL INDEXES;
```

Moving and/or Merging Older Partitions to a Low Cost Storage Tier Device

Using a `MOVE` or `MERGE` operation as part of an Information lifecycle Management strategy, you can relocate older partitions to the most cost-effective storage tier. The data is available for `SELECT` but not for DML operations during the operation. Local indexes are maintained and you will most likely relocate those as part of the merge or move operation as well. Normal index maintenance will be done for partitioned or non-partitioned global indexes and is fully available for select and DML operations.

The following example shows how to merge the January 2006 and February 2006 partitions in the `orders` table, and store them in a different tablespace. Any local index partitions are also moved to the `ts_low_cost` tablespace as part of this operation. The `UPDATE INDEXES` clause ensures that all indexes remain usable throughout and after the operation without additional rebuilds.

```
ALTER TABLE orders
MERGE PARTITIONS p_2006_jan,p_2006_feb
INTO PARTITION p_before_mar_2006 COMPRESS
TABLESPACE ts_low_cost
UPDATE INDEXES;
```

See Also: Chapter 5, "Using Partitioning for Information Lifecycle Management" for more information about the benefits of partition maintenance operations for Information Lifecycle Management

8

Backing Up and Recovering VLDBs

Backup and recovery is one of the most crucial and important jobs for a DBA to protect business data. As the data store grows larger each year, DBAs are continually challenged to ensure that critical data is backed up and that it can be recovered quickly and easily to meet business needs. Very large databases are unique in that they are large and data may come from a myriad of resources. OLTP and data warehouse systems have some distinct characteristics. Generally the availability considerations for a very large OLTP system are no different from the considerations for a small OLTP system. Assuming a fixed allowed downtime, a large OLTP system requires more hardware resources than a small OLTP system.

This chapter proposes an efficient backup and recovery strategy for very large databases to reduce the overall resources necessary to support backup and recovery by leveraging some of the special characteristics that differentiate data warehouses from OLTP systems. This chapter contains the following topics:

- Data Warehousing
- Oracle Backup and Recovery
- Data Warehouse Backup and Recovery
- The Data Warehouse Recovery Methodology
- Best Practice 1: Use ARCHIVELOG Mode
- Best Practice 2: Use RMAN
- Best Practice 3: Use Block Change Tracking
- Best Practice 4: Use RMAN Multi-Section Backups
- Best Practice 5: Leverage Read-Only Tablespaces
- Best Practice 6: Plan for NOLOGGING Operations in Your Backup/Recovery Strategy
- Best Practice 7: Not All Tablespaces Are Created Equal

Data Warehousing

A data warehouse is a system which is designed to support analysis and decision-making. In a typical enterprise, hundreds or thousands of users may rely on the data warehouse to provide the information to help them understand their business and make better decisions. Therefore, availability is a key requirement for data warehousing. This chapter will address one key aspect of data warehousing availability: the recovery of data after a data loss.

Before looking at the backup and recovery techniques in detail, it is important to discuss specific techniques for backup and recovery of a data warehouse. In particular, one legitimate question might be: why shouldn't a data warehouse's backup and recovery strategy be just like that of every other database system?

A DBA should initially approach the task of data warehouse backup and recovery by applying the same techniques that are used in OLTP systems: the DBA must decide what information to protect and quickly recover when media recovery is required, prioritizing data according to its importance and the degree to which it changes. However, the issue that commonly arises for data warehouses is that an approach that is efficient and cost-effective for a 100 GB OLTP system may not be viable for a 10 TB data warehouse. The backup and recovery may take 100 times longer or require 100 times more storage.

Data Warehouse Characteristics

There are four key differences between data warehouses and OLTP systems that have significant impacts on backup and recovery:

1. A data warehouse is typically much larger than an OLTP system. Data warehouses over 10's of terabytes are not uncommon and the largest data warehouses grow to orders of magnitude larger. Thus, scalability is a particularly important consideration for data warehouse backup and recovery.
2. A data warehouse often has lower availability requirements than an OLTP system. While data warehouses are mission critical, there is also a significant cost associated with the ability to recover multiple terabytes in a few hours compared to recovering in a day. Some organizations may determine that in the unlikely event of a failure requiring the recovery of a significant portion of the data warehouse, they may tolerate an outage of a day or more if they can save significant expenditures in backup hardware and storage.
3. A data warehouse is typically updated via a controlled process called the ETL (Extract, Transform, Load) process, unlike in OLTP systems where end-users are modifying data themselves. Because the data modifications are done in a controlled process, the updates to a data warehouse are often known and reproducible from sources other than redo logs.
4. A data warehouse contains historical information, and often, significant portions of the older data in a data warehouse are static. For example, a data warehouse may track five years of historical sales data. While the most recent year of data may still be subject to modifications (due to returns, restatements, and so on), the last four years of data may be entirely static. The advantage of static data is that it does not need to be backed up frequently.

These four characteristics are key considerations when devising a backup and recovery strategy that is optimized for data warehouses.

Oracle Backup and Recovery

In general, backup and recovery refers to the various strategies and procedures involved in protecting your database against data loss and reconstructing the database after any kind of data loss. A backup is a representative copy of data. This copy can include important parts of a database such as the control file, archived redo logs, and data files. A backup protects data from application error and acts as a safeguard against unexpected data loss, by providing a way to restore original data.

Physical Database Structures Used in Recovering Data

Before you begin to think seriously about a backup and recovery strategy, the physical data structures relevant for backup and recovery operations must be identified. The files and other structures that make up an Oracle database store data and safeguard it against possible failures. Three basic components are required for the recovery of an Oracle database:

- Datafiles
- Redo logs
- Control file

Datafiles

An Oracle database consists of one or more logical storage units called tablespaces. Each tablespace in an Oracle database consists of one or more files called datafiles, which are physical files located on or attached to the host operating system in which Oracle is running.

The data in a database is collectively stored in the datafiles that constitute each tablespace of the database. The simplest Oracle database would have one tablespace, stored in one datafile. Copies of the datafiles of a database are a critical part of any backup strategy. The sheer size of the datafiles is the main challenge from a VLDB backup and recovery perspective.

Redo Logs

Redo logs record all changes made to a database's datafiles. With a complete set of redo logs and an older copy of a datafile, Oracle can reapply the changes recorded in the redo logs to re-create the database at any point between the backup time and the end of the last redo log. Each time data is changed in an Oracle database, that change is recorded in the online redo log first, before it is applied to the datafiles.

An Oracle database requires at least two online redo log groups. In each group there is at least one online redo log member, an individual redo log file where the changes are recorded. At intervals, Oracle rotates through the online redo log groups, storing changes in the current online redo log while the groups not in use can be copied to an archive location, where they are called archived redo logs (or, collectively, the archived redo log). For high availability reasons, production systems should always use multiple online redo members per group, preferably on different storage systems. Preserving the archived redo log is a major part of your backup strategy, as it contains a record of all updates to datafiles. Backup strategies often involve copying the archived redo logs to disk or tape for longer-term storage.

Control Files

The control file contains a crucial record of the physical structures of the database and their status. Several types of information stored in the control file are related to backup and recovery:

- Database information required to recover from crashes or to perform media recovery
- Database structure information, such as datafile details
- Redo log details
- Archived log records
- A record of past RMAN backups

Oracle's datafile recovery process is in part guided by status information in the control file, such as the database checkpoints, current online redo log file, and the datafile header checkpoints for the datafiles. Loss of the control file makes recovery from a data loss much more difficult. The control file should be backed up regularly, to preserve the latest database structural changes, and to simplify recovery.

Backup Type

Backups are divided into physical backups and logical backups:

- Physical backups are backups of the physical files used in storing and recovering your database, such as datafiles, control files, and archived redo logs. Ultimately, every physical backup is a copy of files storing database information to some other location, whether on disk or offline storage, such as tape.
- Logical backups contain logical data (for example, tables or stored procedures) extracted from a database with the Oracle Data Pump (export/import) utilities. The data is stored in a binary file that can be imported into an Oracle database.

Physical backups are the foundation of any backup and recovery strategy. Logical backups are a useful supplement to physical backups in many circumstances but are not sufficient protection against data loss without physical backups.

Reconstructing the contents of all or part of a database from a backup typically involves two phases: retrieving a copy of the datafile from a backup, and reapplying changes to the file since the backup, from the archived and online redo logs, to bring the database to the desired recovery point in time. To restore a datafile or control file from backup is to retrieve the file from the backup location on tape, disk, or other media, and make it available to the Oracle Database. To recover a datafile, is to take a restored copy of the datafile and apply to it the changes recorded in the database's redo logs. To recover a whole database is to perform recovery on each of its datafiles.

Backup Tools

Oracle provides the following tools to manage backup and recovery of Oracle databases. Each tool gives you a choice of several basic methods for making backups. The methods include:

- Recovery Manager (RMAN)

  RMAN reduces the administration work associated with your backup strategy by maintaining an extensive record of metadata about all backups and needed recovery-related files. In restore and recovery operations, RMAN uses this information to eliminate the need for the user to identify needed files. RMAN is performant, supporting file multiplexing and parallel streaming, and verifies blocks for physical and (optionally) logical corruptions, on backup and restore.

  Backup activity reports can be generated using `V$BACKUP` views and also through Enterprise Manager.

- Oracle Enterprise Manager

  Enterprise Manager is Oracle's management console that utilizes Recovery Manager for its backup and recovery features. Backup and restore jobs can be intuitively set up and run, with notification of any problems to the user.

- Oracle Data Pump

  Data Pump provides high speed, parallel, bulk data and metadata movement of Oracle database contents. This utility makes logical backups by writing data from an Oracle database to operating system files. This data can later be imported into an Oracle database.

- User-Managed Backups

  The database is backed up manually by executing commands specific to your operating system.

Recovery Manager (RMAN)

Oracle Recovery Manager (RMAN), a command-line and Enterprise Manager-based tool, is the Oracle-preferred method for efficiently backing up and recovering your Oracle database. RMAN is designed to work intimately with the server, providing block-level corruption detection during backup and restore. RMAN optimizes performance and space consumption during backup with file multiplexing and backup set compression, and integrates with leading tape and storage media products via the supplied Media Management Library (MML) API.

RMAN takes care of all underlying database procedures before and after backup or restore, freeing dependency on operating system and SQL*Plus scripts. It provides a common interface for backup tasks across different host operating systems, and offers features not available through user-managed methods, such as data file and tablespace-level backup and recovery, parallelization of backup/recovery data streams, incremental backups, autobackup of control file upon database structural changes, backup retention policy, and detailed history of all backups.

> **See Also:** *Oracle Database Backup and Recovery User's Guide* for more information on RMAN

Oracle Enterprise Manager

Although Recovery Manager is commonly used as a command line utility, Oracle Enterprise Manager enables backup and recovery using a GUI. Oracle Enterprise Manager (EM) supports commonly used Backup and Recovery features:

- Backup Configurations to customize and save commonly used configurations for repeated use
- Backup and Recovery wizards to walk the user through the steps of creating a backup script and submitting it as a scheduled job
- Backup Job Library to save commonly used Backup jobs that can be retrieved and applied to multiple targets
- Backup Job Task to submit any RMAN job using a user-defined RMAN script

Backup Management Enterprise Manager provides the ability to view and perform maintenance against RMAN backups. You can view the RMAN backups, archive logs, control file backups, and image copies. If you select the link on the RMAN backup, then it will display all files that are located in that backup. Extensive statistics about backup jobs, including average throughput, compression ratio, start/end time, and files composing the backup piece can also be viewed from the console.

Oracle Data Pump

Physical backups can be supplemented by using the Data Pump (export/import) utilities to make logical backups of data. Logical backups store information about the schema objects created for a database. Data Pump loads data and metadata into a set of operating system files that can be imported on the same system or moved to another system and imported there.

The dump file set is made up of one or more disk files that contain table data, database object metadata, and control information. The files are written in a binary format. During an import operation, the Data Pump Import utility uses these files to locate each database object in the dump file set.

User-Managed Backups

If you do not want to use Recovery Manager, operating system commands can be used, such as the UNIX `dd` or `tar` commands to make backups. In order to create a user-managed online backup, the database must manually be placed into hot backup mode. Hot backup mode causes additional writes to the online log files, increasing their size.

Backup operations can also be automated by writing scripts. You can make a backup of the entire database at once or back up individual tablespaces, datafiles, control files, or archived logs. An entire database backup can be supplemented with backups of individual tablespaces, datafiles, control files, and archived logs.

Operating system commands or third party backup software can be used to perform database backups. Conversely, the third party software must be used to restore the backups of the database.

Data Warehouse Backup and Recovery

Data warehouse recovery is not any different from an OLTP system. However, a data warehouse may not require all of the data to be recovered from a backup, or in the event of a complete failure, restoration of the entire database before user access can commence. An efficient and fast recovery of a data warehouse begins with a well-planned backup.

The next several sections will help you to identify what data should be backed up and guide you to the method and tools that will allow you to recover critical data in the shortest amount of time.

Recovery Time Objective (RTO)

A Recovery Time Objective, or RTO, is the time duration in which you want to be able to recover your data. Your backup and recovery plan should be designed to meet RTOs your company chooses for its data warehouse. For example, you may determine that 5% of the data must be available within 12 hours, 50% of the data must be available after a complete loss of the Oracle Database within 2 days, and the remainder of the data be available within 5 days. In this case you have two RTOs. Your total RTO is 7.5 days.

To determine what your RTO should be, you must first identify the impact of the data not being available. To establish an RTO, follow these four steps:

1. Analyze and Identify: Understand your recovery readiness, risk areas, and the business costs of unavailable data. In a data warehouse, you should identify critical data that must be recovered in the *n* days after an outage.
2. Design: Transform the recovery requirements into backup and recovery strategies. This can be accomplished by organizing the data into logical relationships and criticality.
3. Build and Integrate: Deploy and integrate the solution into your environment to backup and recover your data. Document the backup and recovery plan.

4. Manage and Evolve: Test your recovery plans at regular intervals. Implement change management processes to refine and update the solution as your data, IT infrastructure, and business processes change.

Recovery Point Objective (RPO)

A Recovery Point Objective, or RPO, is the maximum amount of data that can be lost before causing detrimental harm to the organization. RPO indicates the data loss tolerance of a business process or an organization in general. This data loss is often measured in terms of time, for example, 5 hours or 2 days worth of data loss. A zero RPO means that no committed data should be lost when media loss occurs, while a 24 hour RPO can tolerate a day's worth of data loss.

More Data Means a Longer Backup Window

The most obvious characteristic of the data warehouse is the size of the database. This can be upwards of 100's of terabytes. Hardware is the limiting factor to a fast backup and recovery. However, today's tape storage continues to evolve to accommodate the amount of data that may need to be offloaded to tape (for example, advent of Virtual Tape Libraries which utilize disk internally with the standard tape access interface). RMAN can fully utilize, in parallel, all available tape devices to maximize backup and recovery performance.

Essentially, the time required to back up a large database can be derived from the minimum throughput among: production disk, HBA/network to tape devices, and tape drive streaming specifications * the number of tape drives. The host CPU can also be a limiting factor to overall backup performance, if RMAN backup encryption or compression is used. Backup and recovery windows can be adjusted to fit any business requirements, given adequate hardware resources.

Divide and Conquer

In a data warehouse, there may be times when the database is not being fully utilized. While this window of time may be several contiguous hours, it is not enough to backup the entire database. Therefore, you may want to consider breaking up the database backup over a number of days. RMAN allows you to specify how long a given backup job is allowed to run. When using `BACKUP ... DURATION`, you can choose between running the backup to completion as quickly as possible and running it more slowly to minimize the load the backup may impose on your database.

In the following example, RMAN will backup all database files that have not been backed up in the last 7 days first, run for 4 hours, and read the blocks as fast as possible.

```
BACKUP DATABASE NOT BACKED UP SINCE 'sysdate - 7' PARTIAL DURATION 4:00 MINIMIZE
TIME;
```

Each time this RMAN command is run, it will backup the datafiles that have not been backed up in the last 7 days first. You do not need to manually specify the tablespaces or datafiles to be backed up each night. Over the course of several days, all of your database files will be backed up.

While this is a simplistic approach to database backup, it is easy to implement and provides more flexibility in backing up large amounts of data. Do note that in case of a recovery, RMAN may point you to multiple different storage devices in order to perform the restore. As a result, your recovery time may be longer.

The Data Warehouse Recovery Methodology

Devising a backup and recovery strategy can be a daunting task. When you have 100's of terabytes of data that must be protected and recovered in the case of a failure, the strategy can be very complex. The remainder of this chapter contains several best practices that can be implemented to ease the administration of backup and recovery.

Best Practice 1: Use ARCHIVELOG Mode

Archived redo logs are crucial for recovery when no data can be lost, since they constitute a record of changes to the database. Oracle can be run in either of two modes:

- `ARCHIVELOG`

  Oracle archives the filled online redo log files before reusing them in the cycle.

- `NOARCHIVELOG`

  Oracle does not archive the filled online redo log files before reusing them in the cycle.

Running the database in `ARCHIVELOG` mode has the following benefits:

- The database can be completely recovered from both instance and media failure.
- Backups can be performed while the database is open and available for use.
- Oracle supports multiplexed archive logs to avoid any possible single point of failure on the archive logs.
- More recovery options are available, such as the ability to perform tablespace point-in-time recovery (TSPITR).
- Archived redo logs can be transmitted and applied to the physical standby database, which is an exact replica of the primary database.

Running the database in `NOARCHIVELOG` mode has the following consequences:

- The database can only be backed up while it is completely closed after a clean shutdown.
- Typically, the only media recovery option is to restore the whole database to the point-in-time in which the full or incremental backups were made, which can result in the loss of recent transactions.

Is Downtime Acceptable?

Oracle Database backups can be made while the database is open or closed. Planned downtime of the database can be disruptive to operations, especially in global enterprises that support users in multiple time zones, up to 24-hours per day. In these cases, it is important to design a backup plan to minimize database interruptions.

Depending on the business, some enterprises can afford downtime. If the overall business strategy requires little or no downtime, then the backup strategy should implement an online backup. The database needs never to be taken down for a backup. An online backup requires the database to be in `ARCHIVELOG` mode.

Given the size of a data warehouse (and consequently the amount of time to back up a data warehouse), it is generally not viable to make an offline backup of a data warehouse, which would be necessitated if one were using `NOARCHIVELOG` mode.

Best Practice 2: Use RMAN

Many data warehouses, which were developed on earlier releases of the Oracle Database, may not have integrated RMAN for backup and recovery. However, just as there is a preponderance of reasons to leverage ARCHIVELOG mode, there is a similarly compelling list of reasons to adopt RMAN. Consider the following:

1. Trouble-free backup and recovery
2. Corrupt block detection
3. Archive log validation and management
4. Block Media Recovery (BMR)
5. Easily integrates with Media Managers
6. Backup and restore optimization
7. Backup and restore validation
8. Downtime free backups
9. Incremental backups
10. Extensive reporting

Best Practice 3: Use Block Change Tracking

Enabling block change tracking allows incremental backups to be completed faster, by only reading and writing the changed blocks since the last full or incremental backup. For data warehouses, this can be extremely helpful if the database typically undergoes a low to medium percentage of changes.

See Also: *Oracle Database Backup and Recovery User's Guide* for more information on block change tracking

Best Practice 4: Use RMAN Multi-Section Backups

With the advent of bigfile tablespaces, data warehouses have the opportunity to consolidate a large number of datafiles into fewer, better managed data files. For backing up very large datafiles, RMAN provides multi-section backups as a way to parallelize the backup operation within the file itself, such that sections of a file are backed up in parallel, rather than backing up on a per-file basis.

For example, a 1 TB data file can be sectioned into 10 100 GB backup pieces, with each section backed up in parallel, rather than the entire 1 TB file backed up at once. Thus, the overall backup time for large datafiles can be dramatically reduced.

See Also: *Oracle Database Backup and Recovery User's Guide* for more information on configuring multi-section backups

Best Practice 5: Leverage Read-Only Tablespaces

One of the biggest issues facing a data warehouse is the sheer size of a typical data warehouse. Even with powerful backup hardware, backups may still take several hours. Thus, one important consideration in improving backup performance is minimizing the amount of data to be backed up. Read-only tablespaces are the simplest mechanism to reduce the amount of data to be backed up in a data warehouse. Even with incremental backups, both backup and recovery will be faster if tablespaces are set to read-only.

The advantage of a read-only tablespace is that data only needs to be backed up once. If a data warehouse contains five years of historical data and the first four years of data can be made read-only, then theoretically the regular backup of the database would only back up 20% of the data. This can dramatically reduce the amount of time required to back up the data warehouse.

Most data warehouses store their data in tables that have been range-partitioned by time. In a typical data warehouse, data is generally active for a period ranging anywhere from 30 days to one year. During this period, the historical data can still be updated and changed (for example, a retailer may accept returns up to 30 days beyond the date of purchase, so that sales data records could change during this period). However, once data reaches a certain age, it is often known to be static.

By leveraging partitioning, users can make the static portions of their data read-only. Currently, Oracle supports read-only tablespaces rather than read-only partitions or tables. To take advantage of the read-only tablespaces and reduce the backup window, a strategy of storing constant data partitions in a read-only tablespace should be devised. Here are two strategies for implementing a rolling window.

1. Implement a regularly scheduled process to move partitions from a read-write tablespace to a read-only tablespace when the data matures to the point where it is entirely static.
2. Create a series of tablespaces, each containing a small number of partitions and regularly modify one tablespace from read-write to read-only as the data in that tablespace ages.

One consideration is that backing up data is only half of the recovery process. If you configure a tape system so that it can backup the read-write portions of a data warehouse in 4 hours, the corollary is that a tape system might take 20 hours to recover the database if a complete recovery is necessary when 80% of the database is read-only.

Best Practice 6: Plan for NOLOGGING Operations in Your Backup/Recovery Strategy

In general, one of the highest priorities for a data warehouse is performance. Not only must the data warehouse provide good query performance for online users, but the data warehouse must also be efficient during the ETL process so that large amounts of data can be loaded in the shortest amount of time.

One common optimization leveraged by data warehouses is to execute bulk-data operations using the `NOLOGGING` mode. The database operations which support `NOLOGGING` modes are direct-path loads and inserts, index creation, and table creation. When an operation runs in `NOLOGGING` mode, data is not written to the redo log (or more precisely, only a small set of metadata is written to the redo log). This mode is widely used within data warehouses and can improve the performance of bulk data operations by up to 50%.

However, the tradeoff is that a `NOLOGGING` operation cannot be recovered using conventional recovery mechanisms, since the necessary data to support the recovery was never written to the log file. Moreover, subsequent operations to the data upon which a `NOLOGGING` operation has occurred also cannot be recovered even if those operations were not using `NOLOGGING` mode. Because of the performance gains provided by `NOLOGGING` operations, it is generally recommended that data warehouses utilize `NOLOGGING` mode in their ETL process.

The presence of `NOLOGGING` operations must be taken into account when devising the backup and recovery strategy. When a database is relying on `NOLOGGING` operations, the conventional recovery strategy (of recovering from the latest tape backup and applying the archived logfiles) is no longer applicable because the log files will not be able to recover the `NOLOGGING` operation.

The first principle to remember is, don't make a backup when a `NOLOGGING` operation is occurring. Oracle does not currently enforce this rule, so DBAs must schedule the backup jobs and the ETL jobs such that the `NOLOGGING` operations do not overlap with backup operations.

There are two approaches to backup and recovery in the presence of `NOLOGGING` operations; ETL or incremental backups. If you are not using `NOLOGGING` operations in your data warehouse, then you do not have to choose either of the following options: you can recover your data warehouse using archived logs. However, the following options may offer some performance benefits over an archive log-based approach in the event of recovery. You can also use flashback logs and guaranteed restore points to flashback your database to a previous point in time.

Extract, Transform, and Load

The ETL process uses several Oracle features and a combination of methods to load (re-load) data into a data warehouse. These features consist of:

- Transportable Tablespaces

  Transportable Tablespaces allow users to quickly move a tablespace across Oracle databases. It is the most efficient way to move bulk data between databases. Oracle Database provides the ability to transport tablespaces across platforms. If the source platform and the target platform are of different endianness, then RMAN will convert the tablespace being transported to the target format.

- SQL*Loader

  SQL*Loader loads data from external flat files into tables of an Oracle Database. It has a powerful data parsing engine that puts little limitation on the format of the data in the datafile.

- Data Pump (export/import)

  Oracle Data Pump enables high speed movement of data and metadata from one Oracle database to another. This technology is the basis for Oracle's Data Pump Export and Data Pump Import utilities.

- External Tables

  External Tables is a complement to existing SQL*Loader functionality. It enables you to access data in external sources as if it were in a table in the database. External tables can also be used with the Data Pump driver in order to export data from an Oracle database, using `CREATE TABLE ... AS SELECT * FROM`, and then import data into an Oracle database.

The ETL Strategy

One approach is to take regular database backups and also store the necessary data files to re-create the ETL process for that entire week. In the event where a recovery is necessary, the data warehouse could be recovered from the most recent backup. Then, instead of rolling forward by applying the archived redo logs (as would be done in a conventional recovery scenario), the data warehouse could be rolled forward by re-running the ETL processes. This paradigm assumes that the ETL processes can be easily replayed, which would typically involve storing a set of extract files for each ETL process.

A sample implementation of this approach is to make a backup of the data warehouse every weekend, and then store the necessary files to support the ETL process each night. Thus, at most, 7 days of ETL processing would need to be re-applied in order to recover a database. The data warehouse administrator can easily project the length of time to recover the data warehouse, based upon the recovery speeds from tape and performance data from previous ETL runs.

Essentially, the data warehouse administrator is gaining better performance in the ETL process via `NOLOGGING` operations, at a price of slightly more complex and a less automated recovery process. Many data warehouse administrators have found that this is a desirable trade-off.

One downside to this approach is that the burden is on the data warehouse administrator to track all of the relevant changes that have occurred in the data warehouse. This approach will not capture changes that fall outside of the ETL process. For example, in some data warehouses, end-users may create their own tables and data structures. Those changes will be lost in the event of a recovery.

This restriction needs to be conveyed to the end-users. Alternatively, one could also mandate that end-users create all private database objects in a separate tablespace, and during recovery, the DBA could recover this tablespace using conventional recovery while recovering the rest of the database using the approach of replaying the ETL process.

Incremental Backup

A more automated backup and recovery strategy in the presence of `NOLOGGING` operations leverages RMAN's incremental backup capability. Incremental backups provide the capability to backup only the changed blocks since the previous backup. Incremental backups of datafiles capture data changes on a block-by-block basis, rather than requiring the backup of all used blocks in a datafile. The resulting backup sets are generally smaller and more efficient than full datafile backups, unless every block in the datafile is change.

When you enable block change tracking, Oracle tracks the physical location of all database changes. RMAN automatically uses the change tracking file to determine which blocks need to be read during an incremental backup. The block change tracking file is approximately 1/30000 of the total size of the database.

See Also: *Oracle Database Backup and Recovery User's Guide* for more information on block change tracking and how to enable it

The Incremental Approach

A typical backup and recovery strategy using this approach is to backup the data warehouse every weekend, and then take incremental backups of the data warehouse every night following the completion of the ETL process. Note that incremental backups, like conventional backups, must not be run concurrently with `NOLOGGING` operations. In order to recover the data warehouse, the database backup would be restored, and then each night's incremental backups would be re-applied.

Although the `NOLOGGING` operations were not captured in the archivelogs, the data from the `NOLOGGING` operations is present in the incremental backups. Moreover, unlike the previous approach, this backup and recovery strategy can be completely managed using RMAN.

Flashback Database and Guaranteed Restore Points

Flashback Database is a fast, continuous point-in-time recovery method to repair widespread logical errors. Flashback Database relies on additional logging, called flashback logs, which are created in the Flash Recovery Area and retained for a user-defined period of time according to the recovery needs. These logs track the original block images when they are updated.

When a Flashback Database operation is executed, just the block images corresponding to the changed data are restored and recovered, versus traditional data file restore where all blocks from the backup need to be restored before recovery can start. Flashback logs are created proportionally to redo logs.

For very large and active databases, it may be infeasible to keep all needed flashback logs for continuous point-in-time recovery. However, there may be a need to create a specific point-in-time snapshot (for example, right before a nightly batch job) in the event of logical errors during the batch run. For this scenario, guaranteed restore points (GRP) can be created without enabling flashback logging.

When the GRP is created, flashback logs are maintained just to satisfy flashback database to the GRP and no other point in time, thus saving space. For example, a GRP can be created followed by a nologging batch job. As long as there are no prior nologging operations within the last hour of the creation time of the GRP, flashback database to the GRP will undo the nologging batch job. To flash back to a time after the nologging batch job finishes, then create the GRP at least one hour away from the end of the batch job.

Estimating flashback log space for GRP in this scenario depends on how much of the database will change over the number of days you intend to keep GRP. For example, to keep a GRP for 2 days and you expect 100 GB of the database to change, then plan for 100 GB for the flashback logs. Note that the 100 GB refers to the subset of the database changed after the GRP is created and not the frequency of changes.

Best Practice 7: Not All Tablespaces Are Created Equal

Not all of the tablespaces in a data warehouse are equally significant from a backup and recovery perspective. DBA's can leverage this information to devise more efficient backup and recovery strategies when necessary. The basic granularity of backup and recovery is a tablespace, so different tablespaces can potentially have different backup and recovery strategies.

On the most basic level, temporary tablespaces never need to be backed up (a rule which RMAN enforces). Moreover, in some data warehouses, there may be tablespaces dedicated to scratch space for end-users to store temporary tables and incremental results. These tablespaces are not explicit temporary tablespaces but are essentially functioning as temporary tablespaces. Depending upon the business requirements, these tablespaces may not need to be backed up and restored; instead, in the case of a loss of these tablespaces, the end-users would re-create their own data objects.

In many data warehouses, some data is more important than other data. For example, the sales data in a data warehouse may be crucial and in a recovery situation this data must be online as soon as possible. But, in the same data warehouse, a table storing clickstream data from the corporate website may be much less mission-critical. The business may tolerate this data being offline for a few days or may even be able to accommodate the loss of several days of clickstream data in the event of a loss of database files. In this scenario, the tablespaces containing sales data must be backed up often, while the tablespaces containing clickstream data need only to be backed up once every week or two.

While the simplest backup and recovery scenario is to treat every tablespace in the database the same, Oracle provides the flexibility for a DBA to devise a backup and recovery scenario for each tablespace as needed.

9 Storage Management for VLDBs

Storage performance in data warehouse environments often translates into I/O throughput (MB/s). For OLTP systems the number of I/O requests Per Second (IOPS) is a key measure for performance.

This chapter discusses storage management for the database files in a VLDB environment only. Non-database files, including the Oracle Database software, are not discussed because management of those files is no different from a non-VLDB database. Therefore, the focus is on the high availability, performance, and manageability aspects of storage systems for VLDB environments.

This chapter contains the following topics:

- High Availability
- Performance
- Scalability and Manageability

Note: Oracle supports the use of database files on raw devices and on file systems, and supports the use of Automatic Storage Manager (ASM) on top of raw devices or logical volumes. Oracle recommends that ASM be used whenever possible.

High Availability

High availability can be achieved by implementing storage redundancy. In storage terms, these are mirroring techniques. There are three options for mirroring in a database environment:

- Hardware-based mirroring
- Using ASM for mirroring
- Software-based mirroring not using ASM

Oracle does not recommend software-based mirroring that is not using ASM. The following sections discuss hardware mirroring and mirroring using ASM.

Note: In a cluster configuration, the software you use must support cluster capabilities. ASM is a cluster file system for Oracle database files.

Hardware-Based Mirroring

Most external storage devices provide support for different RAID (Redundant Array of Inexpensive Disks) levels. The most commonly used high availability hardware RAID levels in VLDB environments are RAID 1 and RAID 5. Though less commonly used in VLDB environments, other high availability RAID levels can also be used.

RAID 1 Mirroring

RAID 1 is a basic mirroring technique. Every storage block that is written to storage will be stored twice on different physical devices as defined by the RAID setup. RAID 1 provides fault tolerance because if one device fails, then there is another, mirrored, device that can respond to the request for data. The two writes in a RAID 1 setup are generated at the storage level. RAID 1 requires at least two physical disks to be effective.

Storage devices generally provide capabilities to read either the primary or the mirror in case a request comes in, which may result in better performance compared to other RAID configurations designed for high availability. RAID 1 is the simplest hardware high availability implementation but will require double the amount of storage needed to store the data. RAID 1 is often combined with RAID 0 (striping) in RAID 0+1 configurations. In the simplest RAID 0+1 configuration, individual stripes are mirrored across two physical devices.

RAID 5 Mirroring

RAID 5 requires at least 3 storage devices, but commonly 4 to 6 devices are used in a RAID 5 group. When using RAID 5, for every data block written to a device, parity is calculated and stored on a different device. On reads, the parity is checked. The parity calculation takes place in the storage layer. RAID 5 provides high availability because in case of a device failure, the device's contents can be rebuilt based on the parities stored on other devices.

RAID 5 provides good read performance. Writes may be slowed down by the parity calculation in the storage layer. RAID 5 does not require double the amount of storage but rather a smaller percentage depending on the number of devices in the RAID 5 group. RAID 5 is relatively complex and as a result, not all storage devices support a RAID 5 setup.

Mirroring using ASM

Automatic Storage Manager (ASM) provides software-based mirroring capabilities. ASM provides support for normal redundancy (mirroring) and high redundancy (triple mirroring). ASM also supports the use of external redundancy, in which case ASM will not perform additional mirroring. ASM normal redundancy can be compared to RAID 1 hardware mirroring.

With ASM mirroring, the mirror is produced by the database servers. As a result writes require more I/O throughput when using ASM mirroring compared to using hardware-based mirroring. Depending on your configuration and the speed of the hardware RAID controllers, ASM mirroring or hardware RAID may introduce a bottleneck for data loads.

In ASM, the definition of failure groups enables redundancy, as ASM will mirror data across the boundaries of the failure group. For example, in a VLDB environment, you can define one failure group per disk array, in which case ASM will make sure to place mirrored data on a different disk array. That way, you could not only survive a failure of a single disk in a disk array, but you could even survive the crash of an entire disk array or failure of all channels to that disk array. Hardware RAID configurations typically do not support this kind of fault tolerance.

ASM using normal redundancy requires double the amount of disk space needed to store the data. High redundancy requires triple the amount of disk space.

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

Performance

In order to achieve the optimum throughput from storage devices, multiple disks must work in parallel. This can be achieved using a technique called striping, which stores data blocks in equi-sized slices (stripes) across multiple devices. Striping enables storage configurations for good performance and throughput.

Optimum storage device performance is a trade-off between seek time and accessing consecutive blocks on disk. In a VLDB environment, a 1 MB stripe size provides a good balance for optimal performance and throughput, both for OLTP systems and data warehouse systems. There are three options for striping in a database environment:

- Hardware-based striping
- Software-based striping using ASM
- Software-based striping not using ASM

It is possible to use a combination of striping techniques but you have to make sure you physically store stripes on different devices in order to get the performance advantages out of striping. From a conceptual perspective, software-based striping not using ASM is very similar to hardware-based striping. The following sections discuss hardware-based striping and striping using ASM.

Note: In a cluster configuration, the software you use must support cluster capabilities. ASM is a cluster file system for Oracle database files.

Hardware-Based Striping

Most external storage devices provide striping capabilities. The most commonly used striping techniques to improve storage performance are RAID 0 and RAID 5.

RAID 0 Striping

RAID 0 requires at least 2 devices to implement. Data blocks written to the devices are split up and alternatively stored across the devices using the stripe size. This technique enables the use of multiple devices and multiple channels to the devices.

RAID 0, despite its RAID name, is not redundant. Loss of a device in a RAID 0 configuration results in data loss, and should always be combined with some redundancy in a mission-critical environment. Database implementations using RAID 0 are often combined with RAID 1, basic mirroring, in RAID 0+1 configurations.

RAID 5 Striping

RAID 5 configurations spread data across the available devices in the raid group using a hardware-specific stripe size. As a result, multiple devices and channels are used to read and write data. Due to its more complex parity calculation, not all storage devices support RAID 5 configurations.

Striping Using ASM

Automatic Storage Manager (ASM) always stripes across all devices presented to it in the context of a disk group. A disk group is a logical storage pool in which you create data files. The default ASM stripe size is 1 MB, which is a good stripe size for a VLDB.

> **See Also:** *Oracle Database Storage Administrator's Guide* for more information about ASM configuration

Oracle recommends that you use disks with the same performance characteristics in a disk group. All disks in a disk group should also be the same size for optimum data distribution and hence optimum performance and throughput. The disk group should span as many physical spindles as possible in order to get the best performance. The disk group configuration for a VLDB does not have to be different from the disk group configuration for a non-VLDB.

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

ASM can be used on top of already striped storage devices. If you use such a configuration, then make sure not to introduce hotspots by defining disk groups that span logical devices which physically may be using the same resource (disk, controller, or channel to disk) rather than other available resources. Always make sure that ASM stripes are distributed equally across all physical devices.

ILM

In an Information Lifecycle Management environment, you cannot use striping across all devices, because all data would then be distributed across all storage pools. In an ILM environment, different storage pools typically have different performance characteristics. Therefore, tablespaces should not span storage pools, and hence data files for the same tablespace should not be stored in multiple storage pools.

Storage in an ILM environment should be configured to use striping across all devices in a storage pool. If you use ASM, then separate disk groups for different storage pools should be created. Using this approach, tablespaces will not store data files in different disk groups. Data can be moved online between tablespaces using partition movement operations in the case of partitioned tables, or using the `DBMS_REDEFINITION` package when the tables are not partitioned.

> **See Also:** Chapter 5, "Using Partitioning for Information Lifecycle Management"

Partition Placement

Partition placement is not a concern if you stripe across all available devices and distribute the load across all available resources. If you cannot stripe data files across all available devices, then consider partition placement to optimize the use of all available resources (physical disk spindles, disk controllers, and channels to disk).

I/O-intensive queries or DML operations should make optimal use of all available resources. Storing database object partitions in specific tablespaces, each of which uses a different set of hardware resources, enables you to use all resources for operations against a single partitioned database object. Make sure that I/O-intensive operations can use all resources by using an appropriate partitioning technique.

Hash partitioning and hash subpartitioning on a unique or almost unique column or set of columns with the number of hash partitions equal to a power of 2 is the only technique likely to result in an even workload distribution when using partition placement to optimize I/O resource utilization. Other partitioning and subpartitioning techniques may yield similar benefits depending on your application.

Bigfile Tablespaces

Oracle Database enables the creation of bigfile tablespaces. A bigfile tablespace consists of a single data or temp file which can be up to 128 TB in size. The use of bigfile tablespaces can significantly reduce the number of data files for your database. Oracle Database 11*g* introduces parallel RMAN backup and restore on single data files.

See Also: *Oracle Database Backup and Recovery User's Guide*

As a result, there is no disadvantage to using bigfile tablespaces and you may choose to use bigfile tablespaces in order to significantly reduce the number of data and temp files.

File allocation is a serial process. If you use automatic allocation for your tables and automatically extensible data files, then a large data load can be impacted by the amount of time it takes to extend the file, regardless of whether you use BigFile tablespaces. However, if you pre-allocate data files and you use multiple data files, then you can spawn multiple processes to add data files concurrently.

Scalability and Manageability

A very important characteristic of a VLDB is its large size. Storage scalability and management is an important factor in a VLDB environment. The large size introduces the following challenges:

- Simple statistics suggest that storage components are more likely to fail because VLDBs use more components.
- A small relative growth in a VLDB may amount to a significant absolute growth, resulting in possibly many devices to be added.
- Despite its size, performance and (often) availability requirements are not different from smaller systems.

The storage configuration you choose should be able to cope with these challenges. Regardless of whether storage is added or removed, deliberately or accidentally, your system should remain in an optimal state from a performance and high availability perspective.

Stripe and Mirror Everything (S.A.M.E.)

The S.A.M.E. (Stripe and Mirror Everything) methodology has been recommended by Oracle for many years and is an approach to optimize high availability, performance, and manageability. In order to simplify the configuration further, a fixed stripe size of 1 MB is recommended in the S.A.M.E. methodology as a good starting point for both OLTP and data warehouse systems. ASM implements the S.A.M.E. methodology and adds automation on top of it.

S.A.M.E. and Manageability

In order to achieve maximum performance, the S.A.M.E. methodology proposes to stripe across as many physical devices as possible. This can be achieved without ASM, but if the storage configuration changes, for example, by adding or removing devices, then the layout of the database files on the devices should change. ASM performs this task automatically in the background. In most non-ASM environments, re-striping is a major task that often involves manual intervention.

In an ILM environment, you apply the S.A.M.E. methodology to every storage pool.

ASM Settings Specific to VLDBs

Configuration of Automatic Storage Manager for VLDBs is not very different from ASM configuration for non-VLDBs. Certain parameter values, such as the memory allocation to the ASM instance, may need a higher value.

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

Oracle Database 11*g*, introduces ASM variable allocation units. Large variable allocation units are beneficial for environments that use large sequential I/Os. VLDBs in general, and large data warehouses in particular, are good candidate environments to take advantage of large allocation units. Allocation units can be set between 1 MB and 64 MB in powers of two (that is, 1, 2, 4, 8, 16, 32, and 64). If your workload contains a significant number of queries scanning large tables, then you should use large ASM allocation units. Use 64 for a very large data warehouse system. Large allocation units also reduce the memory requirements for ASM and improve the ASM startup time.

> **See Also:** *Oracle Database Storage Administrator's Guide* for details on how to set up and configure ASM

Monitoring Database Storage Using Database Control

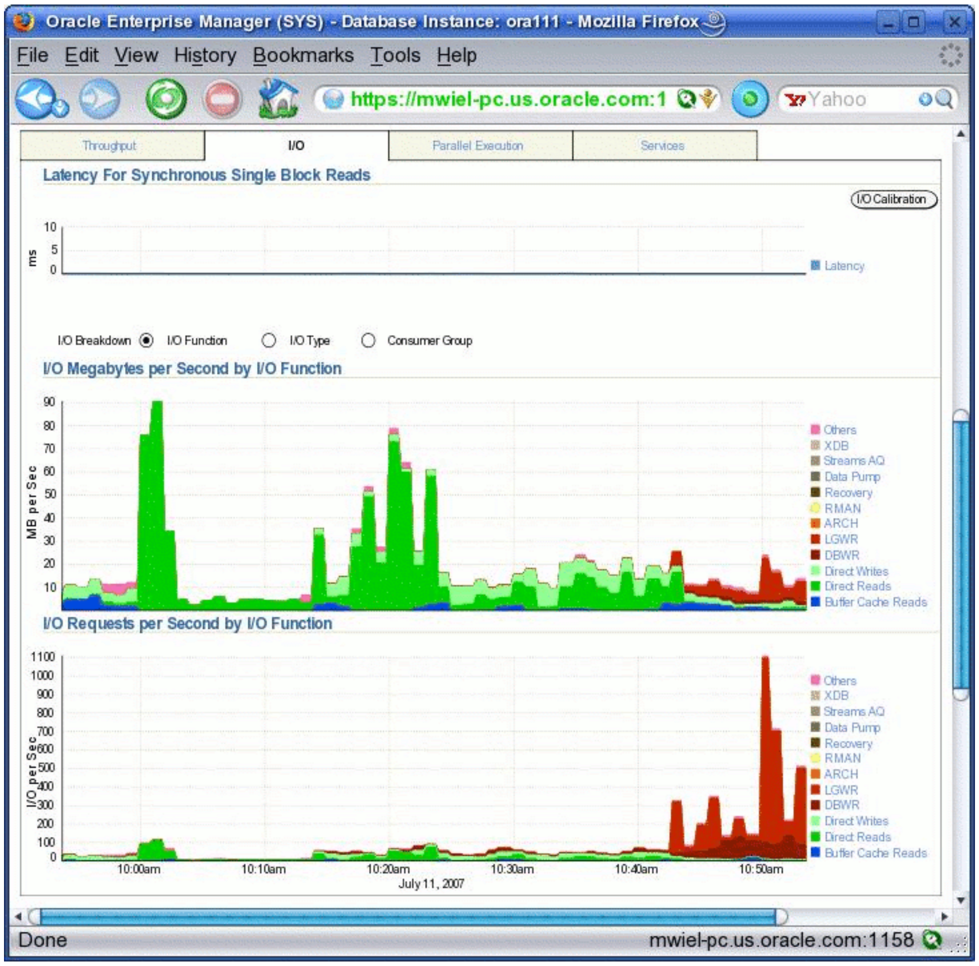
Database Control provides I/O performance overviews. These pages are useful to monitor performance and throughput of the storage configuration. The I/O performance pages can be accessed through the Performance page in Database Control. The main page shows three aspects that define I/O performance:

- Single-block I/O latency: production systems should not show latency of more than ten milliseconds. High latency points to a potential bottleneck in the storage configuration and possibly hotspots.
- I/O megabytes per second: this metric shows the I/O throughput. I/O throughput is an important measure in data warehouse performance.
- I/O per second: this metric, commonly referred to as IOPS, is key in an OLTP application. Large OLTP applications with many concurrent users see a lot of IOPS.

The I/O tab on the main performance page enables breakdown of the I/O overview:

- Function: the function issuing the I/O requests (see Figure 9–1).

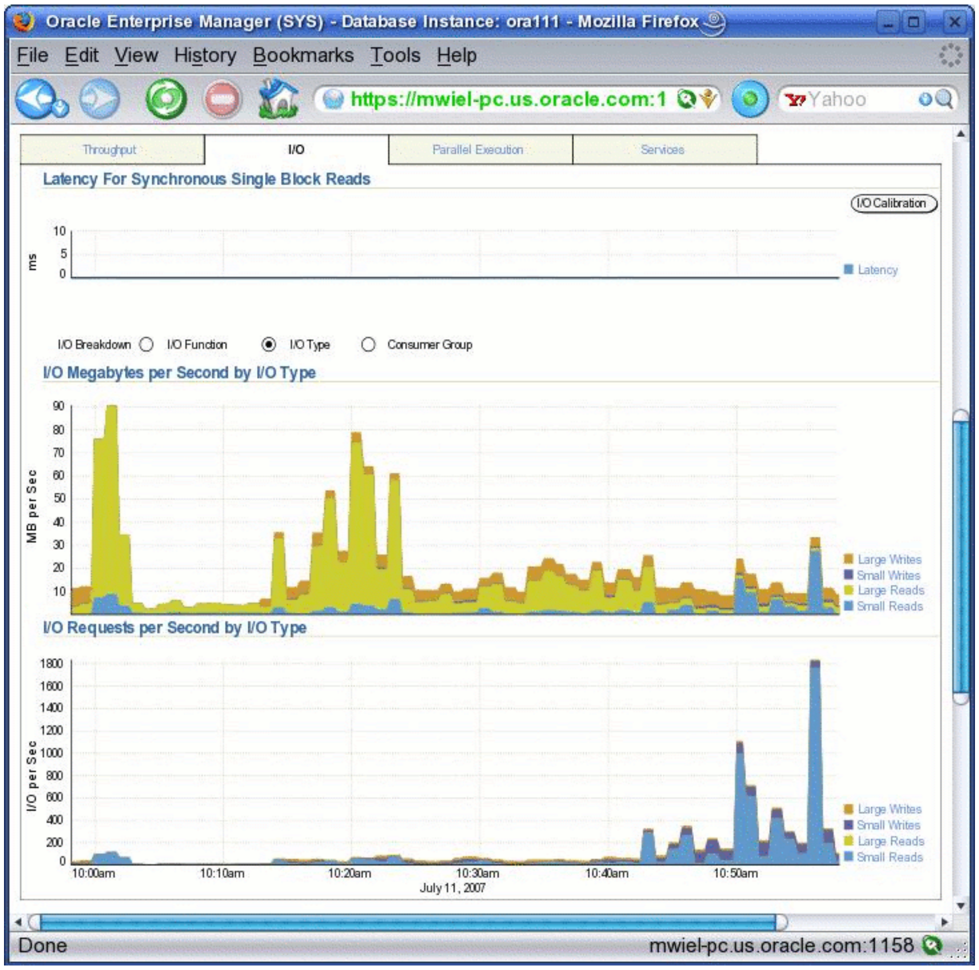
Figure 9–1 System I/O by Function


Oracle Enterprise Manager (SYS) - Database Instance: ora111 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://mwiel-pc.us.oracle.com:1
Yahoo
Throughput
I/O
Parallel Execution
Services
Latency For Synchronous Single Block Reads
I/O Calibration
ms
Latency
I/O Breakdown
I/O Function
I/O Type
Consumer Group
I/O Megabytes per Second by I/O Function
MB per Sec
Others
XDB
Streams AQ
Data Pump
Recovery
RMAN
ARCH
LGWR
DBWR
Direct Writes
Direct Reads
Buffer Cache Reads
I/O Requests per Second by I/O Function
I/O per Sec
10:00am
10:10am
10:20am
July 11, 2007
10:30am
10:40am
10:50am
Done
mwiel-pc.us.oracle.com:1158

I/O Breakdown: [✓ I/O Function] I/O Type Consumer Group

- I/O type: large reads, small reads, large writes, small writes (see Figure 9–2).

Figure 9–2 System I/O by Type of I/O


Oracle Enterprise Manager (SYS) - Database Instance: ora111 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://mwiel-pc.us.oracle.com:1
Yahoo
Throughput
I/O
Parallel Execution
Services
Latency For Synchronous Single Block Reads
I/O Calibration
ms
Latency
I/O Breakdown
I/O Function
I/O Type
Consumer Group
I/O Megabytes per Second by I/O Type
MB per Sec
Large Writes
Small Writes
Large Reads
Small Reads
I/O Requests per Second by I/O Type
I/O per Sec
10:00am
10:10am
10:20am
10:30am
10:40am
10:50am
July 11, 2007
Done
mwiel-pc.us.oracle.com:1158

- Consumer Group: if you configured Oracle Database Resource Manager consumer groups, then you can view I/O performance per group.

You can drill down on the I/O performance views to get an overview of all metrics on a dedicated I/O performance page.

Index