Oracle9*i* Database: Implement Partitioning

Student Guide

D13287GC10
Production 1.0
February 2002
D34456

ORACLE®

Authors

Michael Möller
Jim Womack

Technical Contributors and Reviewers

Herman Baer
Joel Goodman
Stefan Lindbald
Magnus Isakkson
Jean-François Verrier
Alex Melidis
Ananth Raghavan

Publisher

Shane Mattimoe

Contents

Preface

Profile

Prerequisites

- Oracle9*i* Database Administration Fundamentals I (D11321GC11)
- Oracle9*i* Database Administration Fundamentals II (D11297GC11)

Suggested Prerequisites

- Oracle9*i* Database Performance Tuning (D11299GC11)

Suggested Next Course

- Oracle9*i*: Data Warehouse Administration (D13289GC10)

How This Course Is Organized

This is an instructor-led course featuring lecture and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Reference Material

- Oracle9*i* SQL Reference [A90125-01]
- Oracle9*i* Database Reference [A90190-02]
- Oracle9*i* Supplied PL/SQL Packages and Types Reference [A89852-02]

Suggested Reading

- Oracle9*i* Database Administrator's Guide, chapter 17 [A90117-01]
- Oracle9*i* Database Concepts, chapter 12 [A88856-02]
- Oracle9*i* Data Warehousing Guide, chapter 5 [A90237-01]

Oracle Publications

- System release bulletins
- Installation and user's guides
- *read.me* files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

1

Introduction to Partitioning

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- Describe the partitioning architecture, uses, and advantages
- Describe the partition types supported by Oracle RDBMS

ORACLE

1-2

Lesson Content

This lesson will address generalities and the basic functionality of partitioning in Oracle9*i*. Specific syntax, specifications and limitations is covered in upcoming lessons.

VLDB Manageability and Performance Constraints

- Table availability:
  - Large tables are more vulnerable to disk failure.
  - It is too costly to have a large table inaccessible for hours due to recovery.
- Large table manageability:
  - They take too long to be loaded.
  - Indexes take too long to be built.
  - Partial deletes take hours, even days.
- Performance considerations:
  - Large table and large index scans are costly.
  - Scanning a subset improves performance.

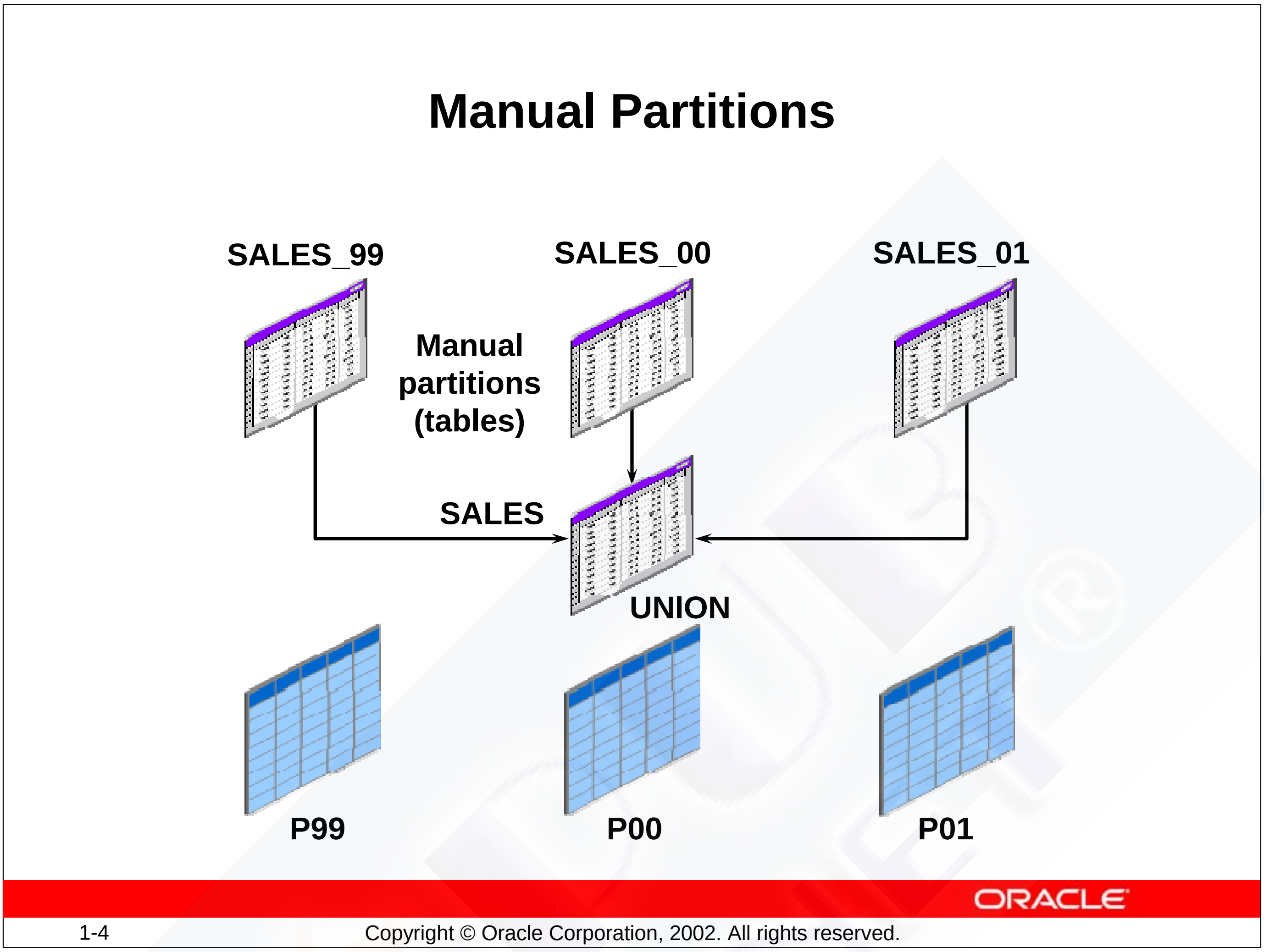
ORACLE

1-3

What Is a VLDB?

A VLDB is a very large database that contains hundreds of gigabytes or even terabytes of data. VLDBs typically owe their size to a few very large tables and indexes rather than a very large number of objects. Below are some typical situations that make it hard to work with VLDBs:

- A disk failure renders a big table inaccessible. The table may be striped over many disks. Users may still need to access the subset of rows unaffected by disk failure.
- Reloading or rebuilding large tables and indexes can greatly exceed any of the company’s downtime allowances.
- In a data warehouse environment, users might query the most recent data more intensely than older data. It would be advantageous to tune the database to meet this pattern of behavior.



Manual Methodology

Prior to the introduction of Oracle Partitioning, manageability constraints were addressed by manually splitting up tables into subsets. Union views were used to mimic the overall table. This had some disadvantages:

- Query optimization and tuning was complex.
- Every manual partition (table) had its own metadata definition, making administration cumbersome.
- Overall primary key and unique constraints were hard or impossible to implement.

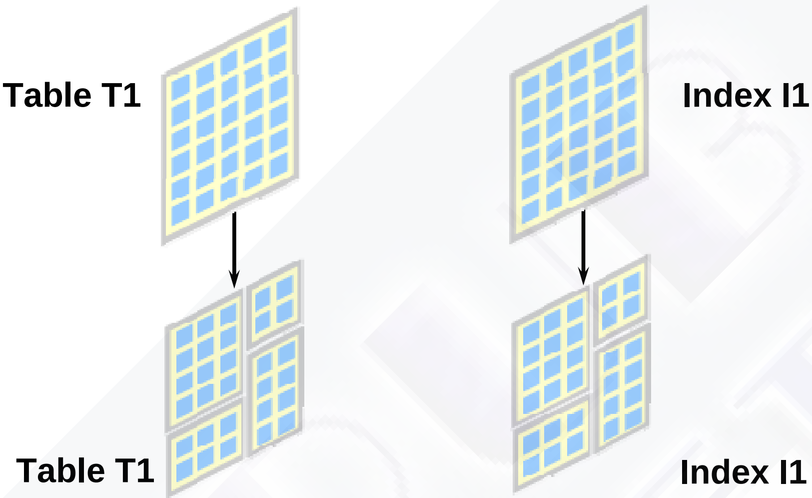
Example:

```
CREATE VIEW accounts AS
      SELECT * FROM accounts_jan00
      UNION ALL
      SELECT * FROM accounts_feb00
      UNION ALL
      ...
     SELECT * FROM accounts_dec00;
```

Partitioning provides a far better way of breaking down tables into manageable pieces.

Partitioned Tables and Indexes

Large tables and indexes can be partitioned into smaller, more manageable pieces.



Partitioned Tables and Indexes

Partitioned tables allow your data to be broken down into smaller, more manageable pieces called partitions, or even subpartitions. Indexes can be partitioned in similar fashion. Each partition can be managed individually, and can function independently of the other partitions, thus providing a structure that can be better tuned for availability and performance.

Partitioning is transparent to existing applications as is standard DML statements run against partitioned tables. However, applications can be enhanced to take advantage of partitioning by using partition-extended table or index names in the application DML.

Benefits of Partitioning: Table Availability

- Partitions can be independently managed.
- Backup and restore operations can be done on individual partitions.
- Partitions that are unavailable do not affect queries or DML operations on other partitions that use the same table or index.

High Availability

Dividing tables and indexes into smaller partitions improves availability of data because if one partition is unavailable, other partitions can be used.

Assume that we have a large table divided into 4 partitions, each residing on a different disk. If recovery must done on one tablespace that holds only the third partition of a total of 4, then partitions 1, 2, and 4 can be accessed simultaneously.

Partitions can also be located in tablespaces that have been made read-only or taken offline. This affects only the partitions in question; all other partitions can still be accessed normally.

Benefits of Partitioning: Large Table Manageability

Oracle provides a variety of methods and commands to manage partitions:

- A partition can be moved from one tablespace to another.
- A partition can be divided at a user-defined value.
- Partitioning can isolate subsets of rows that must be treated individually.
- A partition can be dropped, added, or truncated.
- `SELECT`, `UPDATE`, `INSERT`, and `DELETE` operations can be applied on a partition level instead of a table level.

ORACLE

1-7

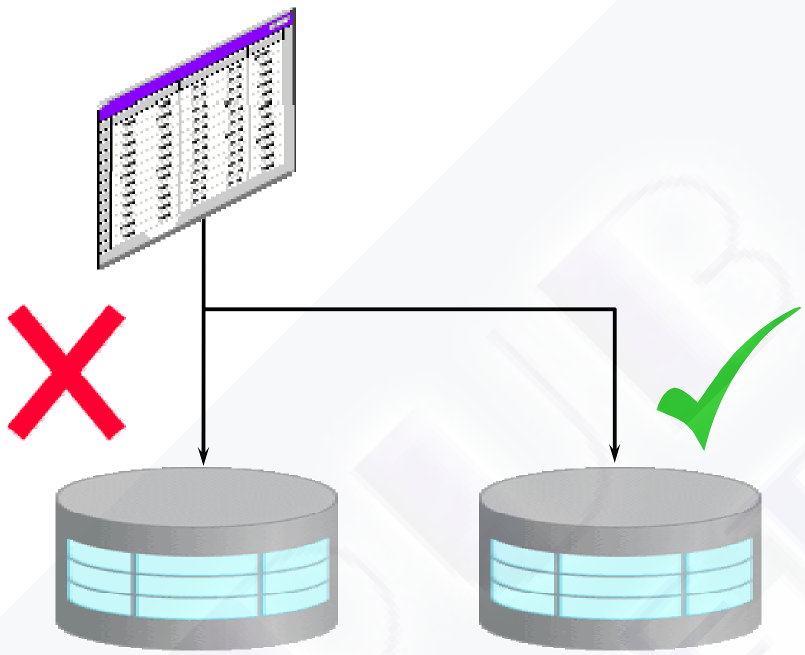
Ease of Administration

Oracle supports many commands for manipulating partitions, for example:

- ALTER TABLE ADD PARTITION
- ALTER TABLE DROP PARTITION (RANGE)
- ALTER TABLE TRUNCATE PARTITION (RANGE)
- ALTER TABLE MOVE PARTITION
- ALTER TABLE SPLIT PARTITION
- ALTER TABLE EXCHANGE PARTITION
- Supporting commands for partition indexes

All of these commands are covered in later lessons.

Manageability: Relocate Table Data



Moving Table Data

Prior to partitioning, it was only possible to relocate table data in a record-by-record fashion, unless you used a direct load method. Now there are several ways to move sets of table rows online.

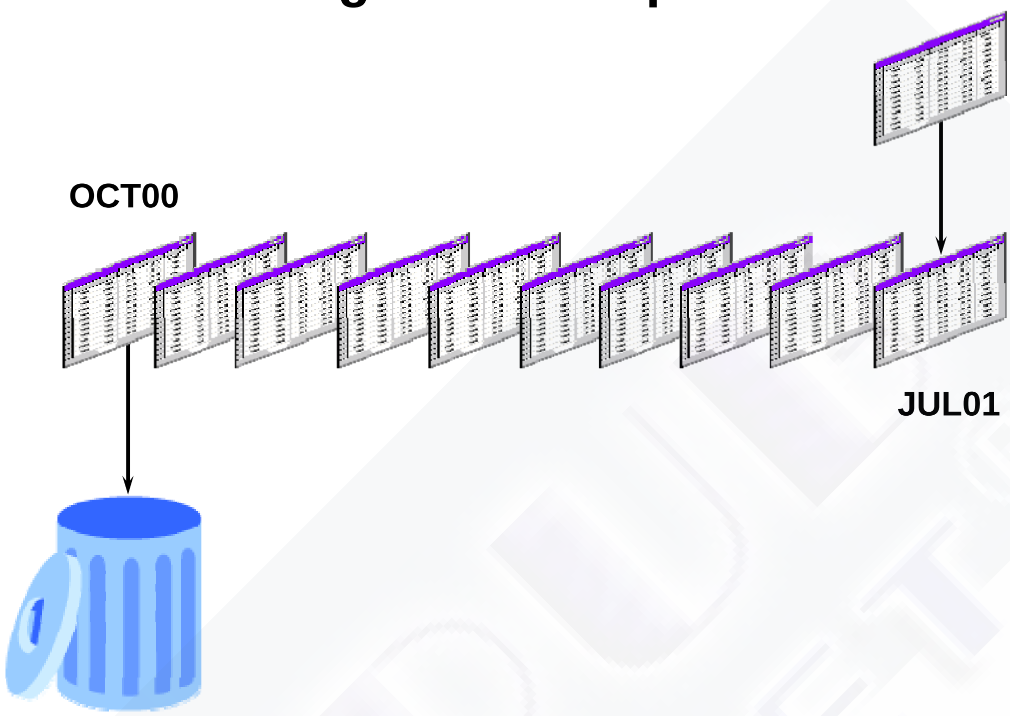
It is possible to physically move:

- A table partition from one tablespace to another.
- A table partition from one database to another.

You can also:

- Logically convert a table partition into a table.
- Logically convert a table into a table partition.
- Modify physical attributes of a table partition.

Manageability: Rolling Window Operations

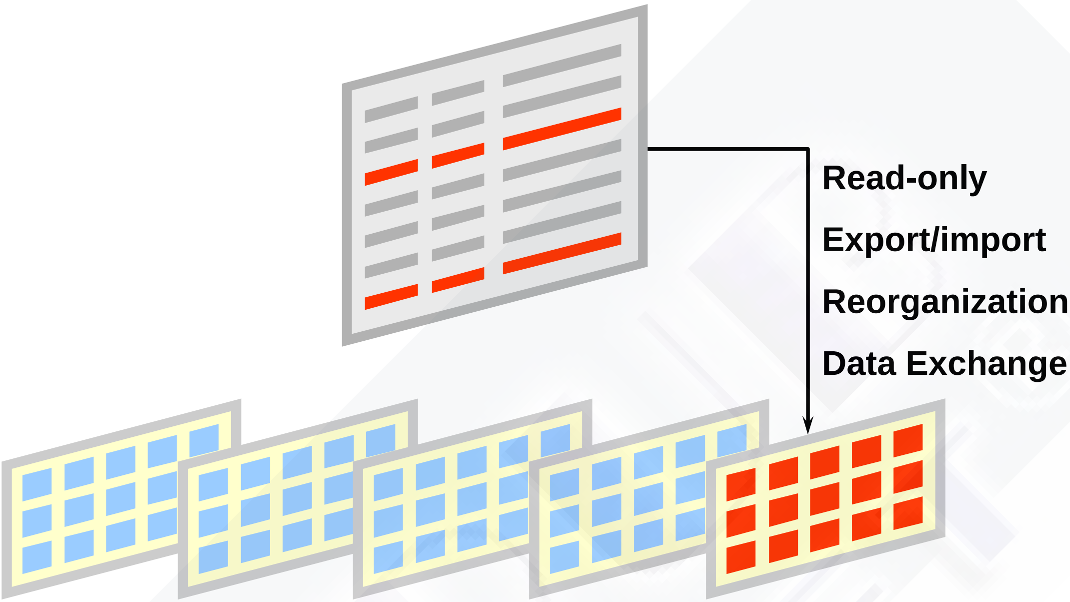


Timeline Databases

Rolling window tables and tables that grow in a linear fashion along business-relevant periods are probably the best example for using partitions:

- Partitions turn costly deleting of individual rows into simple dictionary operations.
- Adding 10,000 rows to a table can be as simple as adding the extents of an already loaded table to an existing partitioned table, thus converting them to a table partition. Again, this is just a dictionary operation.
- Index management can be automated. Purging October 2000 table partition entries and all relevant index entries can be done with one ALTER TABLE statement.
- Creating index entries for the July 2001 table can be done without affecting the existing index entries to the rest of the table.
- DBAs can use SELECT, INSERT, UPDATE, and DELETE on individual partitions.

Manageability: Clearly Defined Record Subsets



Record Subsets

Partitions help isolate any subset of rows that has to be treated individually without affecting the rest of the table:

- Read-only record subsets can be isolated and put into read-only tablespaces.
- Subsets of rows can be easily exported without index or table scanning.
- Subsets of rows can be easily imported without affecting access to the rest of the table.
- Subsets of rows can be reorganized individually, again without affecting the rest of the table.
- Subsets of rows can be converted into a table using a simple dictionary operation.

Benefits of Partitioning: Performance Considerations

- The optimizer eliminates (prunes) partitions that do not need to be scanned.
- Partitions can be scanned, updated, inserted, or deleted in parallel.
- Join operations can be optimized to join "by the partition".
- Partitions can be load-balanced across physical devices.
- Large tables within Real Application Clusters environments can be partitioned.

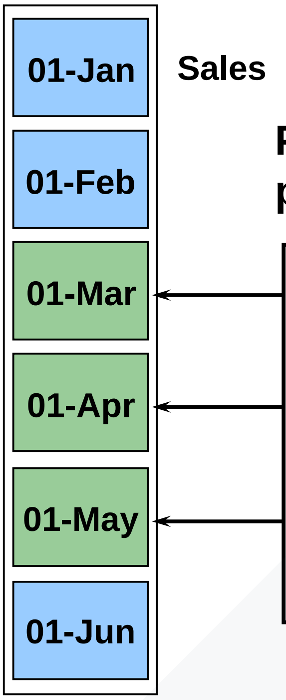
1-11

Improved Performance

The optimizer is aware of the following points when accessing a partitioned table or index:

- If WHERE clauses are specified in a SQL statement, the optimizer can evaluate the statement and based on values, prune partitions that do not need to be accessed.
- Queries and DML operations are narrowed down to partition-level instead of full table/index scan.
- Partition-wise joins are used when the tables are partitioned by the join key. This speeds the join operation, because the amount of data exchanged between query slaves is reduced.
- When the optimizer does sorting, it can apply to partitions instead of to the whole table, causing less temporary sort area in most cases.
- Users can map different partitions to different tablespaces, allowing frequently accessed data to reside on the fastest disks.
- Bulk and maintenance operations can be applied to smaller units of storage.
- Oracle Real Application Clusters can enforce ownership of data by a specific node.

Performance Consideration: Partition Pruning



Partition pruning: only the relevant partitions are accessed

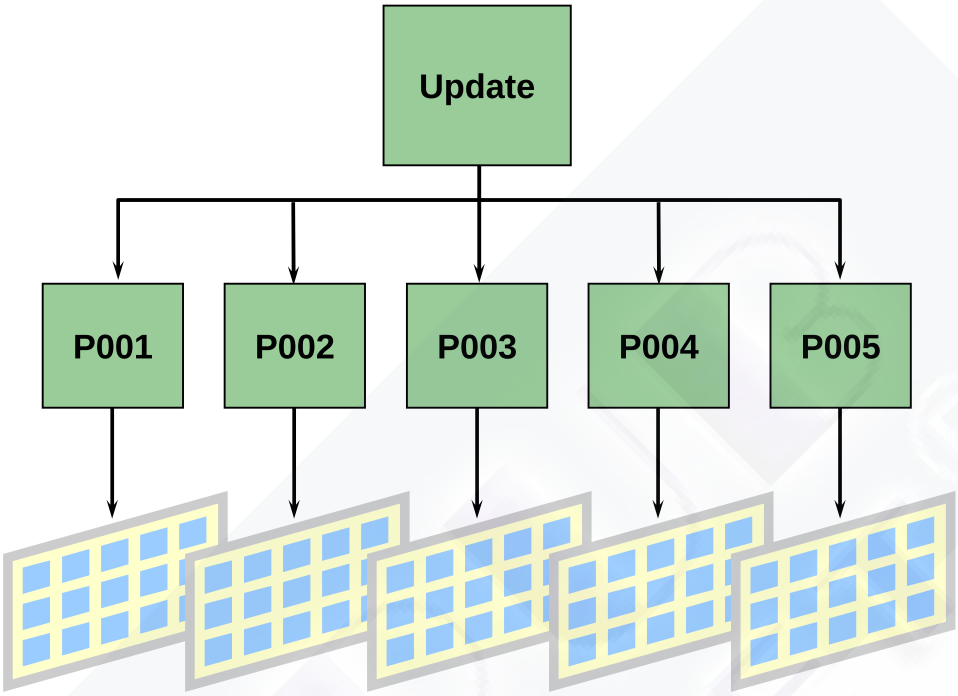
```
SQL> SELECT SUM(amount_sold)
  2  FROM sales
  3  WHERE time_id BETWEEN
  4  TO_DATE('01-MAR-2000',
  5          'DD-MON-YYYY') AND
  6  TO_DATE('31-MAY-2000',
  7          'DD-MON-YYYY');
```


Partition Pruning

Depending on the SQL statement, the Oracle server can explicitly recognize partitions and subpartitions that need to be accessed and the ones that can be eliminated. This optimization is called partition pruning. This can result in substantial improvements in query performance. However, the optimizer cannot prune partitions if the SQL statement applies a function to the partitioning column.

Pruning is expressed using a range of partitions, and the relevant partitions for the query are all the partitions between the first and the last partition of that range. This allows pruning for conjunctive predicates such as c > 10 and c < 20 but not for disjunctive predicates such as c in (10,30) or (c > 10 and c < :B1) or (c > :B2 and c < 1000).

Performance Consideration: Parallel DML



1-13

Parallel DML

Parallelizing DML activities allows for more efficient CPU allocation thus cutting down on the elapsed time for the operation. More row operations can run at the same time without causing contention.

Performance Consideration: Device Load Balancing

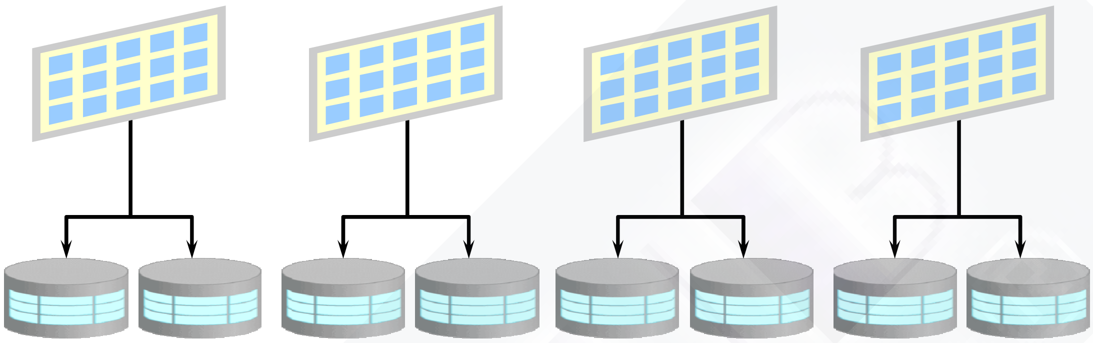
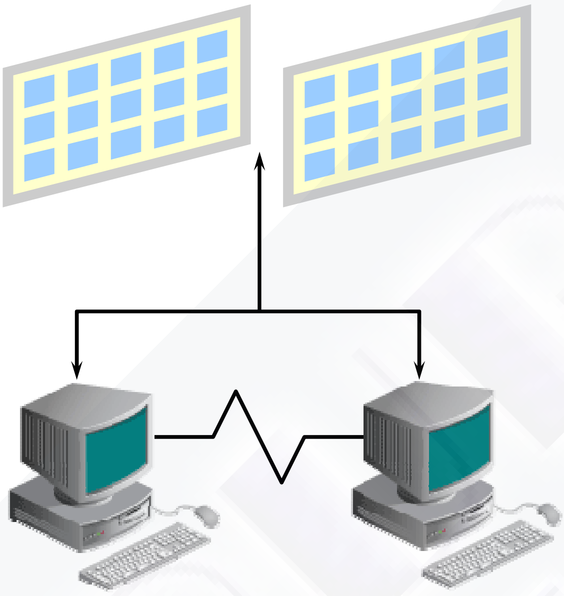


Table Striping using Partitions

With older versions of Oracle, it was hard to stripe a table evenly across disks. It was possible to stripe the initial load across several files in a tablespace. Unfortunately interactive inserts could not be distributed across disks. In OLTP environments it is often crucial to allow for many inserts at peak activity time. Breaking up your target tables into partitions allows you to avoid bottlenecks.

The illustration above shows a configuration with four partitions, each spread out across two disks. A single disk failure affects just one partition.

Performance Improvement: Real Application Clusters



Real Application Clusters

Proper implementation of partitioning can complement the Real Application Clusters environment. It is important to analyze row usage carefully to choose the best way of segmenting user access and row placement using partitions.

Table Versus Index Partitioning

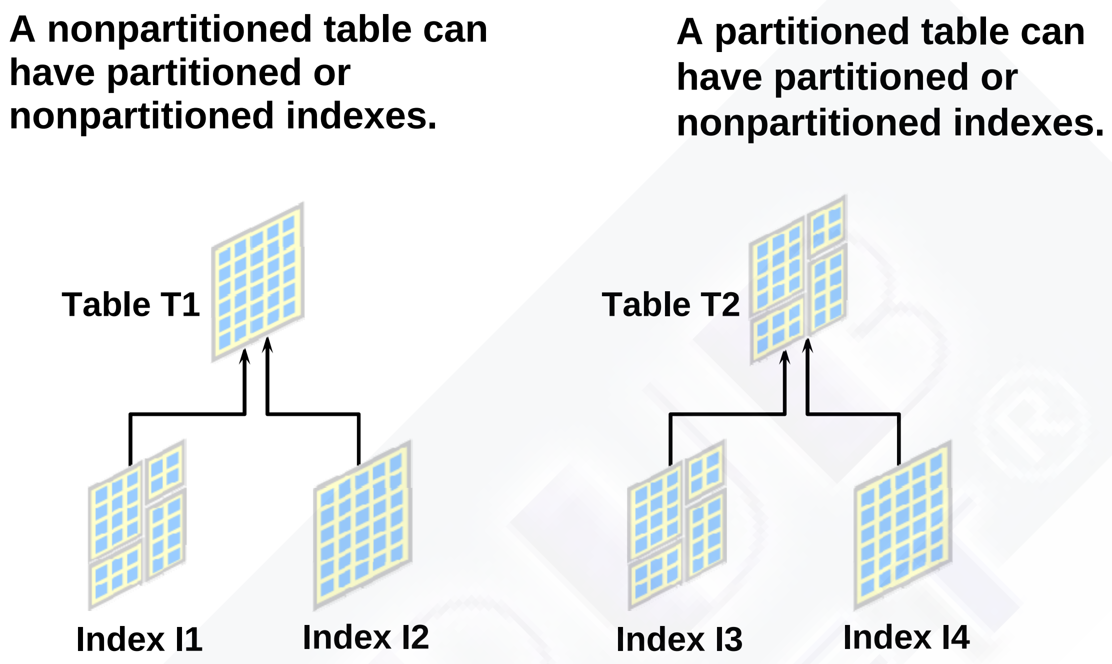


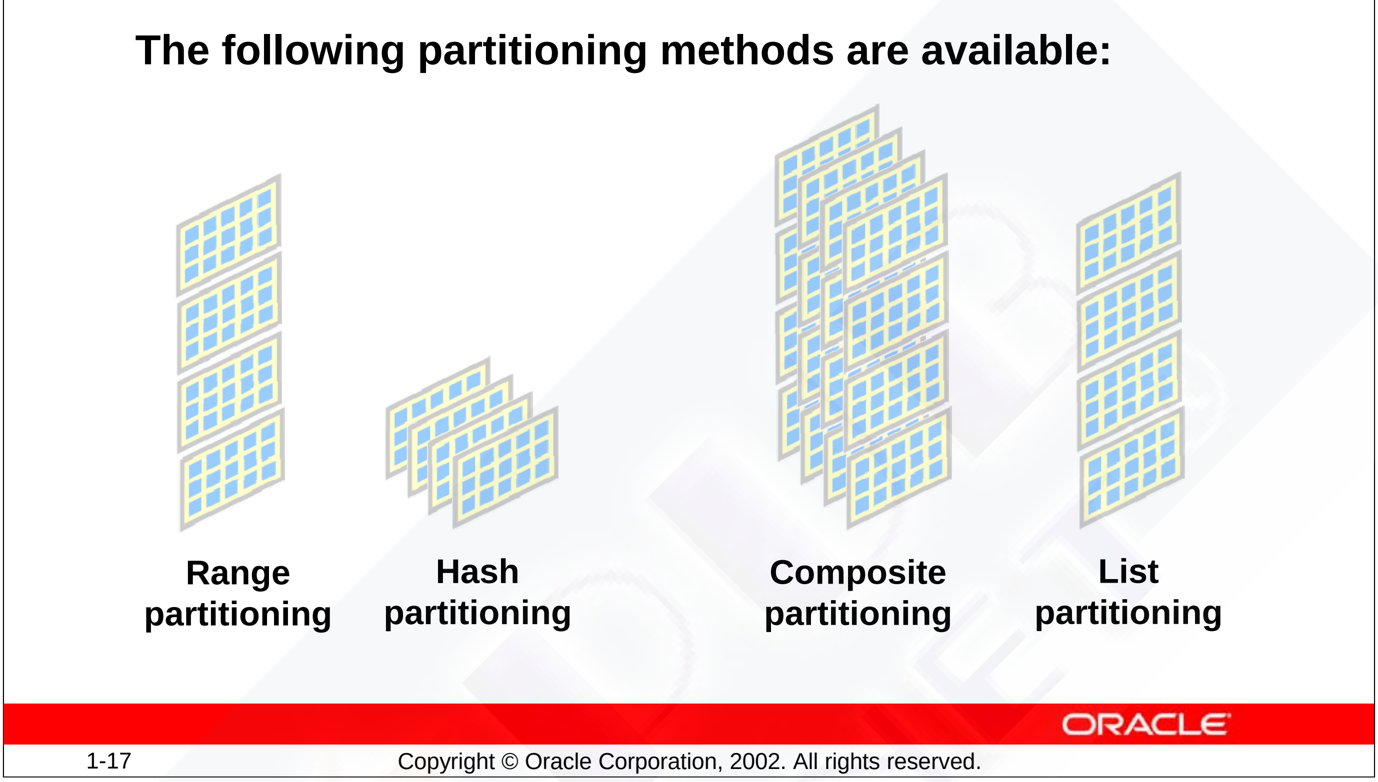
Table and Index Partitioning

In general, you can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables.

- A partitioned table can have partitioned and nonpartitioned indexes.
- A nonpartitioned table can have partitioned and nonpartitioned indexes.
- Bitmap indexes on nonpartitioned tables cannot be partitioned.

However there are design considerations that should be made based on performance, availability, and manageability.

Partitioning Methods



Range Partitioning

Range partitioning uses ranges of column values to map rows to partitions. Range partitions are ordered and this ordering is used to define the lower and upper boundary of a specific partition. Partitioning by range is well suited for historical databases. However, it is not always possible to know beforehand how much data will map into a given range, and in some cases, sizes of partitions may differ quite substantially, resulting in sub-optimal performance for certain operations like parallel DML.

Range partitioning, and partitioning in general, is available in Oracle8 and later versions.

Hash Partitioning

This method uses a hash function on the partitioning columns to stripe data into partitions. It controls the physical placement of data across a fixed number of partitions and gives you a highly tunable method of data placement.

Hash partitioning is available in Oracle8*i* and later versions.

Composite Partitioning

This method partitions data by using the range method and, within each partition, sub-partitions it by using the hash method. This type of partitioning supports historical operations data at the partition level and parallelism (parallel DML) and data placement at the sub-partition level.

Composite partitioning is available in Oracle8*i* and later versions.

List Partitioning

`The LIST` method allows explicit control over how rows map to partitions. This is done by specifying a list of discrete values for the partitioning column in the description for each partition.

`LIST` partitioning is different from `RANGE` partitioning where a range of values is associated with a partition, and from `HASH` partitioning where the user has no control of the row-to-partition mapping. This partition method allows the modeling of data-distributions that follow discrete values that are unordered and unrelated sets of data. These can be grouped and organized together very naturally, using LIST partitioning.

List partitioning is available in Oracle9*i* and later versions.

Partitioned Indexes

- Indexes can be either partitioned or nonpartitioned.
- Choice of indexing strategy allows greater flexibility to suit database and application requirements.
- Indexes can be partitioned with the exception of cluster indexes.
- The same rules apply for indexes as for tables.

ORACLE

1-19

Partitioned Indexes

The rules for partitioning indexes are similar to those for tables. Indexes can be either partitioned or nonpartitioned. Database administrators and application developers need to analyze their indexing needs for their application.

Considerations include the following:

- Type of access to data through the applications
- Performance in accessing data
- Availability in case of disk failure
- Are parallel operations possible?

All of these issues will influence your choice of an indexing strategy.

Verifying Partition Use

- Examining ROWID will confirm the physical placement of the row.
- Examining execution plans will confirm partition pruning.

ORACLE

Verifying Partition Use

The storage of each row in its correct partition can be verified by examining the ROWID. The DBMS_ROWID package is used to decode the file and block number of the row.

The `EXPLAIN PLAN` or other SQL tracing mechanisms can be used to verify that the appropriate partitions are being used in a query or DML. SQL*Plus' AUTOTRACE does not show partition usage.

Proof of Pruning

Proof of partition elimination or pruning may be obtained:

- **By using `tkprof`**
- **Through the explain plan**
- **By setting event 10128**

ORACLE

Proof of Pruning

Pruning is the process wherein the optimizer transparently eliminates partitions from the partition access list. A common example involves sales data, partitioned quarterly. Without table partitioning, you may be required to scan the entire table for dates falling within a particular quarter. With partition pruning, the optimizer will only scan the partition with the relevant range of dates. The partition key does not have to be a date column. The following example uses the explain plan to illustrate partition pruning on a range-partitioned table:

Create the table, four partitions:

```
SQL> create table range_part (col1 number(9))
  2  partition by range (col1)
  3  (partition p1 values less than (10) tablespace system,
  4   partition p2 values less than (20) tablespace system,
  5   partition p3 values less than (30) tablespace users,
  6   partition p4 values less than (MAXVALUE) tablespace users);
```

Insert one row per partition:

```
SQL> insert into range_part values (1);
SQL> insert into range_part values (11);
SQL> insert into range_part values (21);
SQL> insert into range_part values (31);
SQL> commit;
```

Proof of Pruning (continued)

Explain a query that will access a single partition:

```
SQL> EXPLAIN PLAN
  2  set statement_id  = 'range_part'
  3  FOR
  4    SELECT   *
  5      FROM    range_part
  6      WHERE   col1 = 15;
```

Review the explain plan to verify that partition pruning will occur:

```
SQL> SELECT LPAD(' ', 2*(LEVEL-1))||operation operation,
  2  options || '(' || object_name || ')' options, position,
  3  PARTITION_START "START", PARTITION_STOP "STOP"
  4  FROM plan_table
  5  START WITH id = 0 AND statement_id = 'range_part'
  6  CONNECT BY PRIOR id = parent_id AND statement_id = 'range_part'
  7  /

OPERATION        OPTIONS             POSITION START STOP
---------------- ---------------- ---------- ----- ----
SELECT STATEMENT ()                       1
TABLE ACCESS FULL      (RANGE_PART)       1 2      2
```

Pruning proof using `tkprof`:

```
SQL> alter session set sql_trace = true;

SQL> SELECT   *
  2    FROM   range_part
  3   WHERE   col1 = 15;

SQL> !tkprof *.trc 1ist.out explain=sys/change_on_install
```

Partial tkprof output:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  TABLE ACCESS FULL RANGE_PART PARTITION: START=2 STOP=2

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   TABLE ACCESS (FULL) OF 'RANGE_PART' PARTITION: START=2 STOP=2
```

SQL*Loader and Partitioned Objects

- You can load a partitioned table through the conventional path.
- You can sequentially load a partitioned table through the direct path.
- You can parallel load a single table partition through the direct path.

ORACLE

1-23

SQL*Loader Partitioned Object Support

SQL*Loader can load the following:

- A single partition or subpartition of a partitioned table. This can be done by specifying the partition- or subpartition-extended table name in the INTO TABLE clause
- All partitions of a partitioned table. No new syntax is needed.

SQL*Loader Partitioned Object Support in All Paths (Modes)

- Conventional Path: Changed minimally as far back as Oracle7, because mapping a row to a partition or subpartition is handled transparently by SQL.
- Direct Path: When loading a direct path in a single partition, consider the following items:
  - Local indexes can be maintained by the load.
  - Global indexes cannot be maintained by the load.
- Parallel Direct Path: When loading a parallel direct path in a single partition, consider that neither local or global indexes can be maintained by the load.

Parallel direct path loads are used for intrasegment parallelism. Intersegment parallelism can be achieved by concurrent single partition direct path loads, with each load session loading a different partition of the same table.

Summary

In this lesson, you should have learned how to:

- Describe the partitioning architecture, uses, and advantages
- Describe the partition types supported by Oracle RDBMS

ORACLE

Practice Overview: Identifying Partitioning Benefits

This written practice covers the following topics:

- Advantages of Oracle partitioning over manual partitioning
- Benefits to the database administrator when partitioning large tables and indexes
- Partitioning pruning concepts

ORACLE

1-25

Written Exercises

This lesson has no practices, but a few review questions.

2

Implementing Partitioned Tables

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- Describe the partitioning types
- List all of the options for creating a partitioned table
- Create partitioned tables
- Use the data dictionary to verify the partitioned table structure

ORACLE

The CREATE TABLE Statement with Partitioning

An example:

```
SQL> CREATE TABLE simple
  2   ( idx NUMBER, txt VARCHAR2(20) )
  3   PARTITION BY RANGE ( idx )
  4   (   PARTITION VALUES LESS THAN ( 0 )
  5          TABLESPACE data01
  6   ,   PARTITION VALUES LESS THAN ( MAXVALUE )
  7   ) ;
```


Creating a Partitioned Table

This is a simple example. It is a range-partitioned table, in which all rows that contain a negative number in the `idx` column are stored in the first partition, which is stored in the `DATA01` tablespace. All other rows, including those with `NULL` in the `idx` column, are stored in the other partition that is stored in the users default tablespace. Queries on this table will have the benefit of partition pruning and other partition-related performance improvements, if appropriate.

General Syntax

A partitioned table declaration contains three elements:

- The logical structure of the table
- The partition structure, which defines the type and columns
- The structure of each table partition, which has two parts:
  - The logical bounds
  - The physical storage attributes

Version Notes

Real partitioning was not available before Oracle8. Later Oracle7 versions supported Partition Views.

Logical and Physical Attributes

Logical attributes:

- **Normal table structure (columns, constraints)**
- **Partition type**
- **Keys and values**
- **Row movement**

Physical attributes:

- **Tablespace**
- **Extent sizes, block attributes**

Logical and Physical Attributes

When specifying a partitioned table or index, the single statement can specify several attributes. These can easily be divided into those attributes that declare something logical about the table or partition, and those that specify something about the physical storage or manipulation of the table partitions.

Generally, the logical attributes pertain to the table as a whole and will be declared first, while the physical attributes pertain to each partition. Physical storage attributes declared on the table are used as the default values for the partitions.

Normal table structure

There is no change in the way the normal structure of the table is declared when the table is partitioned. The partition clause can simply be appended to an existing table creation script.

Partitioned tables can also be created with the `AS SELECT` clause.

Storage clauses

The storage clauses that can be applied to each table partition are the same storage clauses that can be applied to a normal table, such as `TABLESPACE`, `STORAGE (INITIAL`, `NEXT`), and `PCTFREE`. Therefore, storage clause functionality is not explained further in this course.

Partitioning Type

The Partitioning type is declared in the `PARTITION` clause.

```
SQL> CREATE TABLE ( … column … )
  2  PARTITION BY RANGE ( column_list )
  3  ( PARTITION specifications ) ;
```

```
  2  PARTITION BY HASH ( column_list )
```

```
  2  PARTITION BY LIST ( column )
```

```
  2  PARTITION BY RANGE ( column_list )
        SUBPARTITION BY HASH ( column_list2 )
```

ORACLE

2-5

Partitioning Types

The four types of partitioning are declared in the PARTITION BY partition clause.

The Composite partitioning is limited to being a RANGE partition on the top level, and HASH partitioning on the sublevel.

Multicolumn Partition Key

The Partition Key can consists of several columns, analogous to composite column indexes, except for list partitions. This will be discussed later.

Table Type

Partitioning can be applied to normal heap organized tables and to Index Organized Tables (IOTs). IOTs cannot be list-partitioned.

Clustered tables cannot be partitioned.

Materialized Views (snapshots) can be partitioned.

Specifying Partition Attributes

Each Partition is specified in a partition value clause.

```
…
  PARTITION simple_p1 VALUES ( 'HIGH', 'MED' )
      TABLESPACE data01 PCTFREE 5
, PARTITION simple_p2 VALUES ( 'LOW' )
      TABLESPACE data02 STORAGE ( INITIAL 1M )
…
```

There can be up to 65535 partitions per table.

ORACLE

Specifying Partition Attributes

The code example fragment shows two partitions being specified in a list-partitioned table. The general structure is a comma separated list:

```
PARTITION name partition-key-value storage-attributes
```

The *Partition Name* is optional. If omitted, the system names it SYS_Pnnnn where nnnn is a unique number. Segment names and partition names are distinct. Partition names must only be unique for the table they belong to.

The *Partition Key Value* specification must correspond in type and number to the partition key definition. These must be literals, and not be dependent on the environment; for example, format masks. To avoid such dependencies, use explicit conversion functions. The precise syntax varies with the partition type.

The *storage attributes* syntax is the same as used on normal tables, and is optional and separate for each partition. Defaults are taken from the table declaration, if they are listed there; otherwise, they are taken from tablespace or server defaults as usual. There is no requirement that a separate tablespace must be declared for each partition, but it is usually useful.

Partition Key Value

- The partition key value must be a literal.
- Constant expressions are not allowed, with the exception of TO_DATE conversion.
- The partition key can consist of up to 16 columns

2-7

Partition Key Values

The partition key values must be literals. Even simple expressions are not allowed, such as:

```
3+5, TO_NUMBER('67'), or ASCII('G')
```

The exception is the TO_DATE conversion function, which is used to specify a date literal, when the partition key is of the DATE type. The purpose is to be able to specify the NLS formatting to interpret the date string:

```
TO_DATE('27-12-2002', 'dd-mm-yyyy', 'nls_calendar=gregorian')
```

If this is not done, the CREATE statement will rely on the NLS environment. Note that the year must be specified with four digits.

Range Partitioning

Specify the columns to be partitioned, and the break values for each partition.

- **Each partition must be defined.**
- **The `MAXVALUE` value includes `NULL` values.**

ORACLE

2-8

Range Partitioning

The Partition Key can be any columns from the table, within the data type restrictions.

The partitions end points are specified with:

```
VALUES LESS THAN ( value-list )
```

for each partition. The value-list must correspond in type and position to the partition key. These values are noninclusive. That is, the partition key of rows in a partition does not include the value listed.

The `MAXVALUE` value allows the greatest possible value, and fits all data types. Conversely, the smallest possible value will be stored in the first partition. If you omit the partition with the `MAXVALUE` bound, then there is an implied check constraint on the column.

`NULL` values are stored in the partition with the `MAXVALUE` end point. `NULL`s are treated as "one greater than the highest possible value." There is no way to specify "`MAXVALUE-1`" or otherwise partition `NULL` values separately. You can specify a NOT NULL constraint on the partition key column(s).

Range Partitioning Example

```
SQL> CREATE TABLE simple
  2   ( idx NUMBER, txt VARCHAR2(20) )
  3   PCTFREE 20 TABLESPACE data04
  4   PARTITION BY RANGE ( idx )
  5   (  PARTITION VALUES LESS THAN ( 0 )
  6         TABLESPACE data01
  7   ,  PARTITION VALUES LESS THAN ( MAXVALUE )
  8          PCTFREE 5 ) ;
```


Range Partitioning Example

In the example, rows will partition:

- Any nonzero negative value in the first partition
- Zero, any positive and NULL values in the last partition

The partitions are not explicitly named, and will be called `SYS_Pnnnn`. The first partition has defined storage in tablespace `DATA01`, but uses the default `PCTFREE` value (`20`) from the table definition. The second partition will be stored in the `DATA04` tablespace and use the defined `PCTFREE 5` as its storage attribute.

Multicolumn Partitioning

You can specify multiple columns for a composite partitioning key.

- **The order is significant.**
- **The second column will be examined only after the first column values are equal to the limit specification.**

Multiple Column Partitioning

There can only be one partitioning key, but the key can consists of multiple columns. This is analogous to composite key indexing.

When comparing the row values with the partition end points, in order to determine which partition the row should map to, the following is used:

- If the first column *is less than* the first partition key value, the row belongs to that partition. This means the second column may contain NULLs. The partition key value of the second and subsequent columns is simple ignored.
- If the first column *is equal to* the first partition value key, the second column is compared to the second partition value.
  - If the second column *is less than* the first partition key value, the row belongs to that partition.
  - If it is greater or equal to the first partition key, then the third column will be compared, as above. If there is no third column, then the row belongs to the next partition.
  - If there is no higher partition, the row is rejected.

Multicolumn Example

If this is the partition definition,

```
SQL> CREATE TABLE multicol
  2     ( unit  NUMBER(1), subunit CHAR(1) )
  3  PARTITION BY RANGE ( unit, subunit )
  4  (   PARTITION P_2b VALUES LESS THAN (2,'B')
  5  ,   PARTITION P_2c VALUES LESS THAN (2,'C')
  6  ,   PARTITION P_3b VALUES LESS THAN (3,'B')
  7  ,   PARTITION P_4x VALUES LESS THAN (4,'X') );
```

which partition do the rows then go into?

#	Values	#	Values	#	Values
01	1,'A'	05	1,'Z'	09	1,NULL
02	2,'A'	06	2,'B'	10	2,'C'
03	2,'D'	07	2,NULL	11	3,'Z'
04	4,'A'	08	4,'Z'	12	4,NULL

Multicolumn Example

The partitions are named. To determine which row goes into which partition, the block ID in the `rowid` is displayed below:

```
SQL> SELECT DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) BLOCK,
  2         unit, NVL(subunit,'NULL') FROM multicol ;
BLOCK UNIT SUBUNIT
----- ---- --------
  146    1 A
  146    2 A
  146    1 Z
  146    1 NULL
  162    2 B
  178    2 D
  178    2 NULL
  178    2 C
  194    4 A
  194    3 Z
```

The rows 08 and 12 failed to insert with ORA-14400: inserted partition key does not map to any partition.

Multicolumn Example (continued)

Care must be taken when defining a date using other data types and partition on these multiple columns.

```
CREATE TABLE … ( year NUMBER(4), month NUMBER(2),
                 day NUMBER(2) …)
PARTITION BY RANGE ( year, month, day )
(PARTITION VALUES LESS THAN ( 2001, 01, 32 )
, PARTITION VALUES LESS THAN ( 2001, 02, 29 )
, PARTITION VALUES LESS THAN ( 2001, 03, 32 )
  :
```

The expectation is that only valid dates can be entered, but because a row with values (2000,13,88) will be accepted and stored in the lowest partition, an extra CHECK constraint must be defined on the table to disallow that.

The day end value must be one greater than the month end day, because the values are noninclusive.

Defining the partition key as `( day, month, year )` will cause many problems, and be impossible with the partition key values shown.

A better functionality is to partition direct on a column of type DATE.

The partition key definition should use TO_DATE, with a fully specified date and format mask, to avoid any ambiguities. For example:

```
… VALUES LESS THAN ( TO_DATE('20010101','YYYYMMDD' ) …
```

Using a string as the leading column can give unexpected results. Consider if the DBA_SOURCE table were to be partitioned. There are a few DBMS procedures with thousands of source lines, and most have only a few hundred lines. A simple-minded approach might be:

```
PARTITION BY ( NAME, LINE )
( PARTITION DBA_SOURCE_P1
    VALUES LESS THAN ( 'DBMS',1000 )
, PARTITION DBA_SOURCE_P2
    VALUES LESS THAN ( 'DBMS', MAXVALUE )
, PARTITION DBA_SOURCE_P3
    VALUES LESS THAN ( MAXVALUE, MAXVALUE ) )
```

The erroneous expectation here is that the procedures with lines above 1000 have those rows stored in `DBA_SOURCE_P2`. The partition `DBA_SOURCE_P2` will actually not have a single row in it; all DBMS* source will be stored in `DBA_SOURCE_P3`. The problem in the string comparison is that `'DBMRxxxx'` will compare lower than `'DBMS'` for the first partition, and `'DBMS_xxxx'` will compare larger than `'DBMS'` for partition `DBA_SOURCE_P1` and `DBA_SOURCE_P2`. Thus, only a procedure that is called `'DBMS'` exactly will have its lines greater or equal to 1000 placed in partition `DBA_SOURCE_P2`.

A similar problem can arise if you have two `NUMBER` columns and use fractional numbers.

List Partitioning

Specify the column to partition on, and list the values for each partition.

- **Each partition and each value must be defined.**
- **NULL can be specified.**

```
SQL> CREATE TABLE simple
  2     ( idx NUMBER, txt VARCHAR2(20) )
  3   PARTITION BY LIST ( txt )
  4   ( PARTITION s_top VALUES ( 'HIGH', 'MED' )
  5        TABLESPACE data01
  6   , PARTITION s_bot VALUES ( 'LOW', NULL )
  7        TABLESPACE data02
  8   ) ;
```


LIST Partitioning

The Partition Key can be any single column from the table, within the data type restrictions.

The partitions key values are specified with

```
VALUES ( value-list )
```

for each partition. All values of the partition key value for the partition must be listed as literals. There is no "other" values clause. The string comprising the list of values for each partition can be up to 4K bytes. The total number of `partition key values` for all partitions cannot exceed 64K-1.

`NULL` can be specified as a value. Any literal value, or `NULL`, must only appear once.

You cannot list partition IOTs.

Example

In the example, rows will partition:

- Rows with the value of txt either `'HIGH'`, `'MED'` go into the `s_top` partition
- Rows with the value of txt either `'LOW'` or `NULL` go into the `s_bot` partition
- Any other rows are rejected

The partitions are explicitly named and have a specified tablespace.

Hash Partitioning, Named Partitions

Specify the columns to be partitioned, and the number of partitions:

- **Partition may be defined, or just quantified**
- **NULL is placed in the first partition**
- **Number should be power of two**

```
SQL> CREATE TABLE simple
  2   (idx NUMBER, txt VARCHAR2(20) PRIMARY KEY)
  3  ORGANIZATION INDEX
  4  PARTITION BY HASH ( txt )
  5  ( PARTITION s_h1 tablespace data01
  6  , PARTITION s_h2 tablespace data03
  7  ) ;
```


Hash Partitioning

The Partition Key can consist of any columns from the table, within the data type restrictions.

The partitions end points are not specified. Rows are placed in a partition according to hash value derived from the column values.

The hash partitions can be specified with name and tablespace, but with no other attributes. Other storage attributes must thus be defined in the tablespace.

Alternatively, the hash partitions are not specified, but only the quantity (see next page)

`NULL` values are stored in the first partition.

Example

In the example, rows will be “evenly distributed” in all partitions.

The partitions are explicitly named.

Hash Partitioning: Quantity of Partitions

```
SQL> CREATE TABLE simple
  2     ( idx NUMBER, txt VARCHAR2(20) )
  3  PARTITION BY HASH ( idx )
  4     PARTITIONS 4
  5     STORE IN ( data01, data02 ) ;
```

ORACLE

Hash Partitioning - Quantity of Partitions

In this example the hash partitions are not specified, but only the quantity.

The optional `STORE IN` clause defines which tablespaces to use. If there are not enough tablespaces, the partitions are allocated alternatively to the tablespaces listed.

Number of partitions - Power of two

It is recommended that the number of partitions is a power of two value, that is, 2, 4, 8, 16, or 32, and so on. This is recommended, regardless of which two syntax variations are used to define the hash partitions. If the number is not a power of two, the first few partitions will contain disproportionately more rows. This is due to the hash and partitioning algorithm used.

Composite Partitioning

Composite Partitioning is a partitioning of the partitions.

Hash subpartitioning of a Range Partitioned table:

```
SQL> CREATE TABLE simple
  2    ( idx NUMBER, txt VARCHAR2(20) )
  3   PARTITION BY RANGE ( idx )
  4    SUBPARTITION BY HASH ( txt )
  5     SUBPARTITIONS 4 STORE IN (data01, data02)
  6   ( PARTITION ns_lo VALUES LESS THAN ( 0 )
  7   , PARTITION ns_hi VALUES LESS THAN ( 1E99 )
  8   , PARTITION ns_mx
  9              VALUES LESS THAN ( MAXVALUE )
 10       SUBPARTITIONS 2 STORE IN ( data03 ) ) ;
```


Composite Partitioning

Composite partitioning is a hash partitioning of a range partitioned table partition.

The range partition is specified as a normal range partition type. The hash partition under the range partition is specified with the SUBPARTITION clause, but otherwise uses the same syntax as for simple hash partitioning.

The subpartition partition key can be the same or different from the range partition key.

There is no storage clause associated with each range partition, because they are stored as hash subpartitions. You can specify the hash subpartitions for each range partition, thus indirectly giving each range partition different physical attributes.

Example

This example uses the numbered hash partitions variation to specify the subpartitions.

Only rows with `idx` having `NULL` will be in the `ns_mx` subpartitions. (An additional check constraint prohibiting `idx` greater than `1E99` can be added.)

All range partitions are stored in four hash subpartitions in the tablespaces `data01` and `data02`, except the `ns_mx` partition, which only uses two hash subpartitions stored in the tablespace `data03`.

Composite Partitioning: Another Example

```
SQL> CREATE TABLE simple
      ( idx NUMBER, txt VARCHAR2(20) )
     PARTITION BY RANGE ( idx )
      SUBPARTITION BY HASH ( txt )
     ( PARTITION ns_lo VALUES LESS THAN ( 0 )
        ( SUBPARTITION ns_lo1 TABLESPACE data01
        , SUBPARTITION ns_lo2 TABLESPACE data02
        , SUBPARTITION ns_lo3 TABLESPACE data01
        , SUBPARTITION ns_lo4 TABLESPACE data02 )
     , PARTITION ns_hi VALUES LESS THAN ( 1E99 )
        ( SUBPARTITION ns_hi1 TABLESPACE data01
        , SUBPARTITION ns_hi2 TABLESPACE data02 )
     , PARTITION ns_mx
                    VALUES LESS THAN ( MAXVALUE )
         SUBPARTITIONS 2 STORE IN (data03)
      ) ;
```

2-17

Composite Partitioning - another example

This example uses named hash subpartitions, except for the last partitions, which quantify them by number.

Subpartitions are name SYS_SUBPnnnn with nnnn being unique.

Note the parenthesis in the syntax around the SUBPARTITION keyword in the individual partition definition.

Index Organized Table (IOT) Partitioning

- IOTs can be range or hash partitioned.
- The partition key has to be a subset of the IOT primary key

```
SQL> CREATE TABLE simple
  2   (idx NUMBER, txt VARCHAR2(20), id2 NUMBER
  3    , CONSTRAINT s_pk PRIMARY KEY (idx, txt) )
  4   ORGANIZATION INDEX
  5   PARTITION BY HASH ( txt )
  6   ( PARTITION s_h1 tablespace data01
  7   , PARTITION s_h2 tablespace data03
  8   ) ;
```

ORACLE

IOT Partitioning

The partitioning clauses are unchanged for partitioning an Index Organized Table (IOT).

The `INCLUDING` clause of the IOT can only be defined on the table, that is, it must be the same for all partitions.

Segment and Partition Names

An IOT table consists of one or two segments, which are named `SYS_IOT_TOP_nnnn` and `SYS_IOT_OVER_nnnn`. A partitioned IOT has the same segment names, and the partition segments can be named by the system or in the partition clause.

OVERFLOW Segment Partitioning

- **The OVERFLOW segment of an IOT is equipartitioned with the table partitions.**
- **Storage attributes of the OVERFLOW segment are specified for each partition.**

```
  …
PARTITION s_10 VALUES LESS THAN ( 10 )
  TABLESPACE INDX01
  OVERFLOW TABLESPACE DATA04
  …
```

2-19

Partitioning of the OVERFLOW segment of IOT

The index organized table (IOT) can consist of two segments, the table and the OVERFLOW segment. When the IOT is partitioned, the overflow segment is likewise partitioned, creating one overflow partition for every table partition.

An OVERFLOW clause is placed as part of the partition physical attributes. If the OVERFLOW is not specified in the partition clause, the default storage from the table overflow clause is used. The overflow partition is still created.

The OVERFLOW clause must be specified in the table definition, if it is specified in any partition clause. Omitting all OVERFLOW clauses creates a partitioned IOT without overflow. You cannot separately name the overflow partitions, because they receive the same partition name as the table partitions.

OVERFLOW Segment Example

```
SQL> CREATE TABLE simple
  2   (idx NUMBER PRIMARY KEY, txt VARCHAR2(10))
  3  ORGANIZATION INDEX
  4  OVERFLOW TABLESPACE data01
  5  PARTITION BY RANGE ( idx )
  6  ( PARTITION s_10 VALUES LESS THAN ( 10 )
  7      TABLESPACE INDX01
  8      OVERFLOW TABLESPACE DATA04
  9  , PARTITION s_20 VALUES LESS THAN ( 20 )
 10      TABLESPACE INDX02
 11   ) ;
```


Example

If the row has an overflow, it is stored in the overflow partition that is associated with the table partition where the beginning of the row is stored.

The OVERFLOW on line 4 is the table level definition of the overflow segment. On lines 9 and 10 the OVERFLOW clause is missing, therefore the overflow segment partition uses the overflow definitions from line 4.

If you omitted lines 5 onwards, the IOT would be created as a nonpartitioned IOT with an overflow segment.

LOB Partitioning

- LOB segments are equipartitioned with the table partition.
- Storage attributes are specified for each LOB in each partition.

```
   …
 PARTITION s_10 VALUES LESS THAN ( 10 )
    TABLESPACE data01
    LOB ( txt ) STORE AS st_10
        ( DISABLE STORAGE IN ROW
          TABLESPACE data03 )
   …
```

ORACLE

2-21

Partitioning of LOB segments

A table with LOB columns will probably have an additional segment for every LOB (CLOB, NCLOB and BLOB, but not BFILE). When the table is partitioned, the LOB segment is likewise partitioned, creating one LOB partition for every table partition, for each LOB column.

A LOB storage clause can be specified as part of the partition physical attributes. If the LOB storage clause is not specified in the partition clause, the default storage from a possible table level LOB storage clause is used. The LOB partition is still created.

You can name both the LOB segment and partitions. All the LOB attributes can be specified separately for each partition.

You cannot specify a LOB column as part of a partition key.

LOB Segment Example

```
SQL> CREATE TABLE simple
  2     ( idx NUMBER, txt CLOB )
  3  LOB ( txt ) STORE AS s_lob
  4      ( TABLESPACE data04 )
  5  PARTITION BY RANGE ( idx )
  6  ( PARTITION s_10 VALUES LESS THAN ( 10 )
  7      TABLESPACE data01
  8      LOB ( txt ) STORE AS st_10
  9          ( DISABLE STORAGE IN ROW
 10            TABLESPACE data03 )
 11  , PARTITION s_20 VALUES LESS THAN ( 20 )
 12      TABLESPACE data02
 13   ) ;
```


LOB segment Example

Rows are stored in the appropriate partition as explained for previous range partitions. The LOB of the row is stored in the LOB partition that is associated with the table partition where the row is stored.

The LOB definition on lines 3 and 4 is the table level definition of the LOB segment. Around line 12, the LOB clause is missing, therefore the LOB segment partition uses the LOB definition from lines 3 and 4. The LOB partition is named SYS_LOB_Pnnnn.

If you omit lines 5 onwards, the table would be a normal table with one LOB segment.

The table level specification of the LOB segment in lines 3 and 4 is optional.

Partitioned Object Tables and Partitioned Tables with Object Types

- Object tables can be partitioned.
- Tables containing object types can be partitioned.
- Nested Tables cannot be partitioned.

2-23

Partitioned Object Tables and Partitioned Tables with Object Types

Range, hash and composite partitioning are supported.

Attributes that are of type object, REF, or are part of a nested table or VARRAY cannot be part of the partition key.

Global indexes are allowed for range partitioning, and on the range partitions of a composite partitioned table.

If the object identifier is user defined, then some or all of the columns used to define the object identifier can also be used in the partition key, if required. However, the partition key cannot explicitly use an object identifier.

Attributes that are of type object, REF, or are part of a nested table or VARRAY cannot be part of the partition key.

Partitioning tables with VARRAYs are similar to partitioning tables with object type columns.

Updateable Partition Keys

Because performing `UPDATE` on a row alters the value of the columns that define which partition the row belongs to, the following can happen:

- **The update results in the row still being mapped to the same partition.**
- **The update makes the row map to another partition, and therefore is disallowed.**
- **The update makes the row map to another partition, and therefore the row is moved to the new partition.**

Updates on the Partition Value

The first case is allowed. The value of the column of the partition key can be updated.

The choice between the second and third case is controlled by the

```
ROW MOVEMENT DISABLED | ENABLED
```

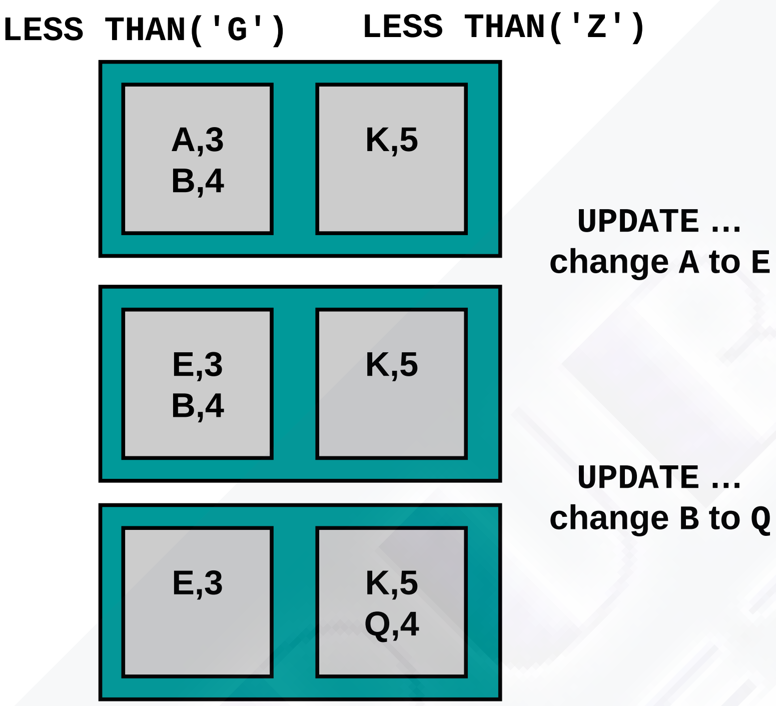
attribute of the table. This is `DISABLED` by default.

When a row moves, all indexes referring to it are maintained. This can generate considerable redo activity. Although this can be thought of as a DELETE from one partition, followed by an INSERT in the other partition, only the UPDATE trigger will fire once for the statement or row as defined. The moved row has a new ROWID.

An `UPDATE` that attempts to alter the partition column values to outside the partition bound values, if row movement is disabled, fails (ORA-14406).

Ordinary row migration can still occur, and will be within the partition.

Row Movement



Row Movement

This illustrates a table with two partitions and three rows.

There are two updates. The first update does not move the row, but the other update requires the row to move.

Row Movement Example

```
SQL> CREATE TABLE simple ( idx NUMBER ... )
  2   ENABLE ROW MOVEMENT
  3 PARTITION BY RANGE ( idx )
  4 ( PARTITION s_neg VALUES LESS THAN ( 0 )
    … ;
SQL> INSERT INTO simple VALUES ( 1, 'Moving' ) ;
SQL> SELECT idx,BLOCK(ROWID),rowid FROM simple ;
     1   181   AAAB/GAADAAAAC1AAA
SQL> UPDATE simple SET idx=0 ;
SQL> SELECT idx,BLOCK(ROWID),rowid FROM simple ;
     0   181   AAAB/GAADAAAAC1AAA
SQL> UPDATE simple SET idx=-1 ;
SQL> SELECT idx,BLOCK(ROWID),rowid FROM simple ;
     -1  117   AAAB/FAADAAAAB1AAA
SQL> ALTER TABLE simple DISABLE ROW MOVEMENT ;
SQL> UPDATE simple SET idx=0
ORA-14402: updating partition key column would
cause a partition change
```


Row Movement Example

The `BLOCK` function above is the `DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid)` function; it shows the block number of the row.

When the column value of the partition key changes from 0 to –1, the row moves position and its rowid changes as well.

The `ALTER TABLE` statement disables the `ROW MOVEMENT`. Updating the row again in order to get a row movement, fails this time with

```
ORA-14402: updating partition key column would cause a
partition change.
```

Equipartitioning

- If two tables have the same partition keys and partition key values, then they are equipartitioned.
- This is useful for tables with a common key, like master-detail relationships.
- Partition-wise join operation requires equipartitioning.
- Indexes can be equipartitioned with the table.

ORACLE

2-27

Equipartitioning

Equipartitioned tables (and indexes) can have different physical attributes for each partition, as long as they have the same partition key definition and the same partition key values in each partition.

This simplifies data management because all rows in both tables are easily manipulated together as partitions correspond.

As seen previously, the OVERFLOW part of an IOT is equipartitioned with the table.

Master-Detail equipartitioning is useful for export and import by the partition; the corresponding rows are kept together.

Equipartitioning a table and its materialized view avoids bulk loading of data that invalidates the whole MV. Query rewrite can occur on other partitions.

Indexes will be covered in the next lesson. Local indexes are always equipartitioned.

Partition Extended Table Names

Specify the partition in a table to limit an operation:

```
SQL> SELECT idx
  2   FROM simple PARTITION ( s_neg ) ;
```

```
SQL> DELETE  FROM simple SUBPARTITION ( s_h2 ) ;
```

```
SQL> CREATE TABLE sim2
  2   AS SELECT * FROM simple PARTITION ( p3 ) ;
```

ORACLE

Specifying a Partition

The optional `PARTITION` or `SUBPARTITION` clause can be used to specify the name of the partition to use. This will limit the operation to the named partition, acting as a `WHERE` clause.

Note that most DDL operations have separate syntax for manipulating a partition, you do not use the Partition Extended Name.

General Restrictions

- Partitioned tables cannot contain `LONG` data.
- All partitions must reside in tablespaces with the same block size.
- You cannot partition on LOBs.
- Comparison of partition keys is done by binary values.

General Restrictions

Any partitions belonging to the table must be of the same blocksize. Other segments, such as overflow or LOB segments do not need the same block size. Indexes are separate objects and are not constrained to the table block size.

While you cannot use LOB as the partition key, LOBs can be part of a partitioned table.

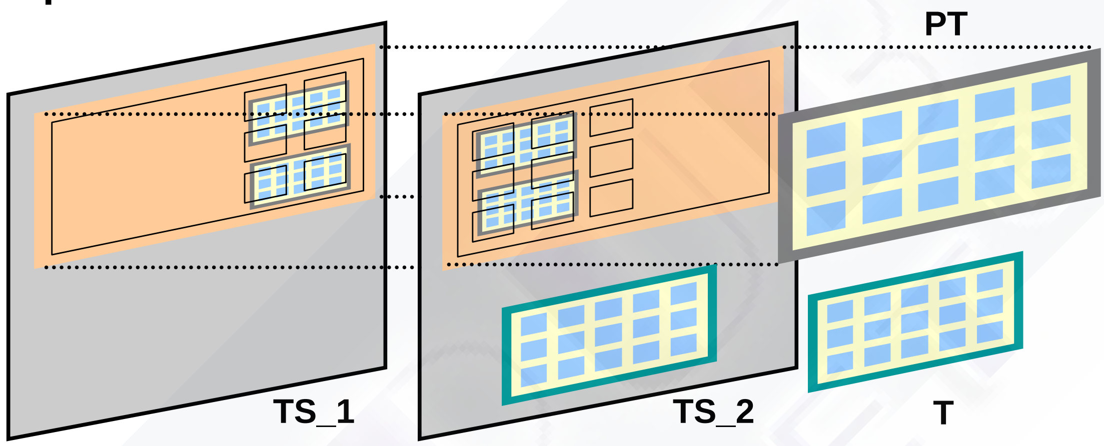
Comparison is done by binary values. This might have consequences in the expected row mapping for strings, because it will not be lexical sorting.

Release 9*i* Release 1 Restrictions

`TIMESTAMP WITH TIME ZONE` cannot be a partition key.

Table, Partition, and Segment Relations

- **A partitioned table is an object consisting of subobjects, the partitions.**
- **The table is “virtual,” and consists of physical partitions.**



Segment or Partition

The Data Dictionary views of the segment of partitioned tables and other objects is not a simple one-to-one relation. For ordinary tables, a table is both an object and a segment. For a partitioned table, the table is just an object, but also consists of partitions, each of which is an object and a segment. The simple relation of a table residing in a tablespace (thus the name), is extended to a table’s partitions residing in different tablespaces.

Shown here is a partitioned table (PT). The table object and partition objects, still called the table segments, are residing in tablespaces TS_1 and TS_2. The table consists of two partitions, that is, two segments called *table partition segments*, in the tablespace TS_1 and TS_2, respectively.

A normal table (T) is shown for comparison with one table segment in tablespace TS_2.

In summary:

- Tablespace TS_1 and TS_2
- Tables T and PT
- Table segments T and PT
- Table partition segment (name not shown on diagram) SYS_P0022, SYS_P0023

Each partition segment can consist of many extents, all in the same tablespace.

Data Dictionary Views
Tables

Name	Purpose	N
DBA_TABLES	Table structure, Partition Y/N	T
DBA_PART_TABLES	Partition type, default values	T
DBA_TAB_*PARTITIONS	Partitions detail	P
DBA_*PART_KEY_COLUMNS	Partition keys	P

*** SUB variation**

T = per Table
P = per Partition

Data Dictionary Views Tables

In the following discussion and slides, the views are consistently given by their `DBA_` prefix version. The `USER_` and `ALL_` versions of the views also exist.

Basic table definition `DBA_TABLES` shows if the table is partitioned (YES, NO) and if row movement is enabled. The `TABLESPACE_NAME` column and other storage attributes are `NULL` if it is partitioned, because the table has no storage, only its partitions. There is one row for each table.

```
SQL> SELECT TABLE_NAME, TABLESPACE_NAME,
  2         PARTITIONED, ROW_MOVEMENT
  3    FROM USER_TABLES ;

TABLE_NAME TABLESPACE_NAME PARTITIONED ROW_MOVE
---------- --------------- ----------- --------
HR_EMP                     YES         ENABLED
MULTICOL                   YES         ENABLED
ORDINARY   USERS           NO
SIMPLE                     YES         DISABLED
```

The partition definition is in `DBA_PART_TABLES` which describes the partition type (range, list, and so on), the partition key, and default storage attributes of partitions (the corresponding fields in `DBA_TABLES` are `NULL`). There is one row for each table.

Data Dictionary Views Tables (continued)

```
SQL> SELECT TABLE_NAME, PARTITIONING_TYPE,
  2    SUBPARTITIONING_TYPE, PARTITION_COUNT,
  3    PARTITIONING_KEY_COUNT,DEF_TABLESPACE_NAME
  4    FROM USER_PART_TABLES ;
TABLE_NAME TYPE   SUBTYPE PAR.CNT PAR.KEY_CNT DEF_TAB.SP
---------- ------ ------- ------- ----------- ----------
COMPOS     RANGE  HASH          3           1 USERS
MULTICOL   RANGE  NONE          4           2 USERS
SIMPLE     LIST   NONE          2           1 USERS
```

The individual partitions are described in `DBA_TAB_PARTITIONS`, which describe the end point (range) or values of the partition, and the storage attributes. There is one row for each partition. The subpartitions are described in `DBA_TAB_SUBPARTITIONS`.

```
SQL> SELECT TABLE_NAME, PARTITION_NAME,
  2    COMPOSITE, SUBPARTITION_COUNT,
  3    PARTITION_POSITION, HIGH_VALUE, TABLESPACE_NAME
  4    FROM USER_TAB_PARTITIONS ;

TABLE_NAME P.NAME COM SUB.CNT PART.POS. HIGH_VALUE TABLESP
---------- ------ --- ------- --------- ---------- -------
SIMPLE     S_BOT  N0        0         2 'LOW', NUL DATA02
SIMPLE     S_TOP  N0        0         1 'HIGH','ME DATA01
MULTICOL   P_2B   N0        0         1 2, 'B'     USERS
MULTICOL   P_2C   N0        0         2 2, 'C'     USERS
MULTICOL   P_3B   N0        0         3 3, 'B'     USERS
MULTICOL   P_4X   N0        0         4 4, 'X'     USERS
COMPOS     NS_HI  YES       2         2 1E99       USERS
COMPOS     NS_LO  YES       4         1 0          USERS
COMPOS     NS_MX  YES       2         3 MAXVALUE   USERS
```

Note the COMPOSITE column value is `'N0 '` (N-zero-space) for “No”.

```
SQL> SELECT TABLE_NAME, PARTITION_NAME,
  2    SUBPARTITION_NAME, SUBPARTITION_POSITION,
  3    TABLESPACE_NAME FROM USER_TAB_SUBPARTITIONS ;

TABLE_NAME PART.NAME  SUBP.NAME  PART.POS. TABLESPACE
---------- ---------- ---------- --------- ----------
SIMPLE     NS_HI      NS_HI1             1 DATA01
SIMPLE     NS_HI      NS_HI2             2 DATA02
SIMPLE     NS_LO      NS_LO1             1 DATA01
```

Data Dictionary Views Tables (continued)

The partition keys are described in `DBA_PART_KEY_COLUMNS` and `DBA_SUBPART_KEY_COLUMNS` for partitions and subpartitions, respectively. There is one row for every column specified in any partition.

```
SQL> SELECT NAME "TABLE_NAME", 'PART' PART, COLUMN_NAME,
         COLUMN_POSITION
       FROM USER_PART_KEY_COLUMNS
       WHERE TRIM(OBJECT_TYPE)='TABLE'
     UNION ALL
     SELECT NAME "TABLE_NAME", 'SUBP' PART, COLUMN_NAME,
         COLUMN_POSITION
       FROM USER_SUBPART_KEY_COLUMNS
       WHERE TRIM(OBJECT_TYPE)='TABLE' ;

TABLE_NAME PART COLUMN_NAME    COL.POS.
---------- ---- ------------ ----------
COMPOS     PART IDX                   1
MULTICOL   PART SUBUNIT               2
MULTICOL   PART UNIT                  1
SIMPLE     PART TXT                   1
COMPOS     SUBP CHR                   2
COMPOS     SUBP TXT                   1
```

These two data dictionary tables contain partition keys for both tables and indexes, thus the `WHERE` clause. The values returned from column `OBJECT_TYPE` are space padded, thus the TRIM function. The two tables have been denormalized for a combined query adding the `PART` column to show from which table the row came.

All listings here have been edited to fit.

Data Dictionary Views Segments

Name	Columns to show
DBA_SEGMENTS	PARTITION_NAME, SEGMENT_TYPE
DBA_EXTENTS	PARTITION_NAME, SEGMENT_TYPE
DBA_OBJECTS	SUBOBJECT_NAME, OBJECT_TYPE

2-34

Data Dictionary Views Segments

The DBA_SEGMENT table has the column PARTITON_NAME to identify the partitions belonging to a *table-segment*. All partitions segments have the segment name of the table in SEGMENT_NAME. PARTITION_NAME is NULL for nonpartitioned tables, otherwise it contains the partition name. The SEGMENT_TYPE column shows if the segment is a partition segment, in addition to the different segment types (data, index, and so on).

The DBA_EXTENTS table also has the PARTITION_NAME column.

The DBA_OBJECTS table refers to the partitions of a table as subobjects. Each of these has its own object ID. Thus, a two-partitioned table has three entries: one for the table, and two partitions. The SUBOBJECT_NAME, OBJECT_TYPE identify this.

Data Dictionary Views Segments (continued)

Example

```
SQL> SELECT SEGMENT_NAME,PARTITION_NAME,SEGMENT_TYPE,
  2    TABLESPACE_NAME FROM USER_SEGMENTS ;

SEGMENT_NAME   PARTITION_NAME SEGMENT_TYPE       TABLESPACE
-------------- -------------- ------------------ ---------
ORDINARY                      TABLE              USERS
SIMPLE         S_BOT          TABLE PARTITION    DATA02
SIMPLE         S_TOP          TABLE PARTITION    DATA01
MULTICOL       P_2B           TABLE PARTITION    USERS
MULTICOL       P_2C           TABLE PARTITION    USERS
MULTICOL       P_3B           TABLE PARTITION    USERS
MULTICOL       P_4X           TABLE PARTITION    USERS
COMPOS         NS_LO1         TABLE SUBPARTITION DATA01
COMPOS         NS_LO2         TABLE SUBPARTITION DATA02
COMPOS         NS_LO3         TABLE SUBPARTITION DATA01
COMPOS         NS_LO4         TABLE SUBPARTITION DATA02
COMPOS         NS_HI1         TABLE SUBPARTITION DATA01
COMPOS         NS_HI2         TABLE SUBPARTITION DATA02
COMPOS         SYS_SUBP456    TABLE SUBPARTITION DATA03
COMPOS         SYS_SUBP457    TABLE SUBPARTITION DATA03

SQL> SELECT OBJECT_NAME, SUBOBJECT_NAME, OBJECT_ID,
  2       DATA_OBJECT_ID, OBJECT_TYPE, STATUS
  3    FROM USER_OBJECTS ;

OBJ._NAME  SUB_NAME O_ID DO_ID OBJECT_TYPE        STATUS
--------- --------- ---- ----- ------------------ -------
COMPOS        NS_HI  6739       TABLE PARTITION    VALID
COMPOS        NS_HI1 6745 6745  TABLE SUBPARTITION VALID
COMPOS        NS_HI2 6746 6746  TABLE SUBPARTITION VALID
COMPOS        NS_LO  6738       TABLE PARTITION    VALID
COMPOS        NS_LO1 6741 6741  TABLE SUBPARTITION VALID
COMPOS        NS_LO2 6742 6742  TABLE SUBPARTITION VALID
…
COMPOS               6737       TABLE              VALID
MULTICOL      P_2B   6721  6721 TABLE PARTITION    VALID
…
MULTICOL             6720       TABLE              VALID
ORDINARY             6544  6544 TABLE              VALID
SIMPLE        S_BOT  6751  6751 TABLE PARTITION    VALID
SIMPLE        S_TOP  6750  6750 TABLE PARTITION    VALID
SIMPLE               6749       TABLE              VALID
```

Summary

In this lesson, you should have learned how to:

- Create the four different partition types
- Specify storage attributes for partitions
- Apply partitioning to tables, IOTs, and tables with LOBs
- Examine the data dictionary to verify how the partitions are defined

ORACLE

Practice Overview: Creating Partitioned Tables

This practice covers the following topics:

- Creating a partitioned table of each type
- Using the data dictionary to verify the partition structure
- Inserting a few records into tables and verifying with ROWID that the records are placed in the expected partitions
- Verifying that partition pruning occurs

ORACLE

2-37

3

Implementing Partitioned Indexes

Objectives

After completing this lesson, you should be able to do the following:

- Describe the table and index partition relationships
- List all the options of partitioned indexes
- Create some partitioned indexes
- Use the data dictionary to verify the partitioned index structure

ORACLE

Partitioned Indexes

- Indexes can be partitioned like tables.
- Partitioned or nonpartitioned indexes can be used with partitioned or nonpartitioned tables.

Partitioned Indexes

The same management benefits and performance improvement that can be achieved by partitioning tables is achieved by partitioning indexes.

When an index is partitioned, each partition is a complete separate index. There is no *master index* to decide which index to use, because that information is inherent in the partition information.

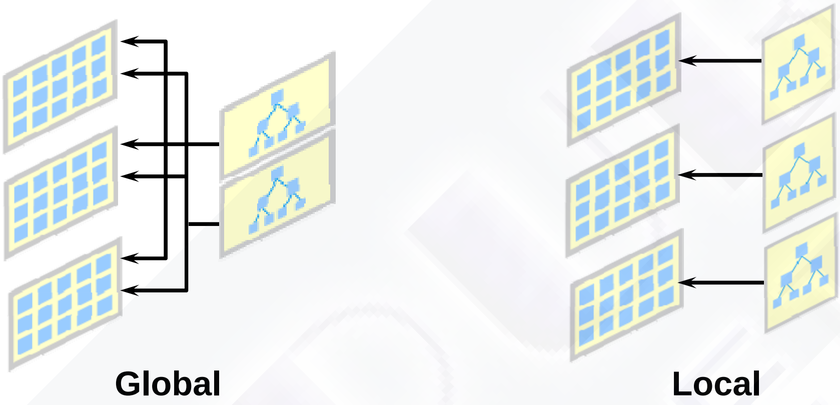
The syntax used to specify the partitioning of an index is very similar to that used to partition a table.

Index partitions can be specified to follow the table, thus greatly simplifying the partition definition.

There are some restrictions on index type (such as bitmap or unique) and index partition type (such as global or local nonprefixed) combinations.

Partitioned Index Attributes: Global or Local

- **The partitions of a Global Partitioned Index are defined independently from the table that it indexes. A Local Partitioned Index corresponds in partitioning to the table.**



Global or Local

Global partitioned indexes can be defined on a nonpartitioned table.

Local indexes can only be defined on partitioned tables.

A local index has the same partitioning key as the table partitioning key.

Local indexes are automatically maintained; that is, changes made on the partitions on the table are automatically repeated on the local indexes. More details are provided in the *Maintenance of Partitions* lesson.

Local Subpartitioned

If the table is composite partitioned, the local index has corresponding subpartitions. There are no local partitions that correspond to the range partitions of the composite partitioned table.

Partitioned Index Attributes: Prefixed or Nonprefixed

In a prefixed index, all leftmost columns of the index key are the same as all the columns in the partition key of the index.

ORACLE

Prefixed or Nonprefixed

The prefixed or nonprefixed attribute is not directly specifiable, but is a consequence of the index key columns and partition key column specification matching.

In a prefixed index, both leading index key and index partition key are the same.

In a nonprefixed index, the leading index key is not the same as the index partition keys.

Index Partitioning Types

- Partitioned indexes can be:
  - Global or local
  - Prefixed or nonprefixed
- Allowed partitioning types are:
  - Global, not equipartitioned, and prefixed
  - Local, equipartitioned, and prefixed
  - Local, equipartitioned, and nonprefixed
- A normal nonpartitioned index is also a "partition type."
- All index types can be partitioned.

ORACLE

Index Partitioning Types

Although the various partition attributes can be combined in more ways, only these three index partition types are supported.

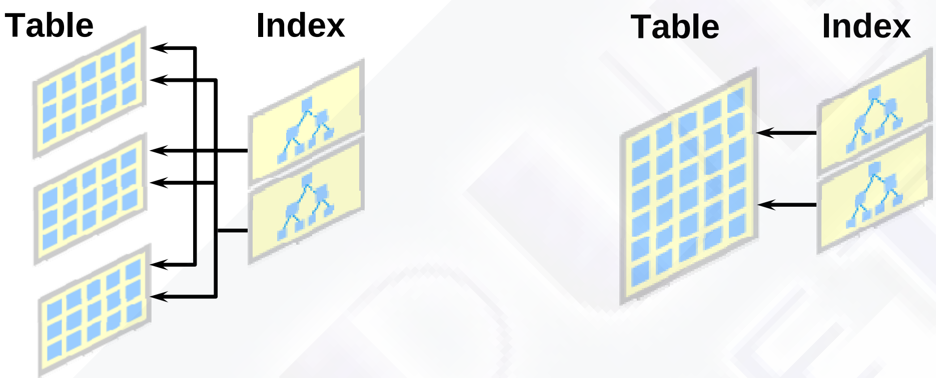
Index Types

Indexes can be of different types; B*Tree, Bitmap, Bitmap Join, and Functional. The index types are independent of the index partition type. All index types can be partitioned, but some restrictions apply. For example, a bitmap index cannot be global partitioned.

Global Indexes

Global indexes:

- **Must be prefixed**
- **Only allow `RANGE` partitioning**
- **Must include `MAXVALUE` on all columns**



Global Prefixed Indexes

Global indexes can be made on plain tables and partitioned tables, as shown above.

There is no required relation between the index partitioning and the table partitioning.

Only B*Tree indexes can be global partitioned.

Global indexes and non-partitioned indexes on partitioned tables require more space in the index for the rowid reference, because it must address any tablespace.

Global indexes can be unique or nonunique.

Global Nonprefixed

It is not possible to create a nonprefixed global index, because there are no management or performance benefits when compared to a nonpartitioned index.

Global Index Example

```
SQL> CREATE INDEX idx ON emp ( first_name )
  2  GLOBAL PARTITION BY RANGE ( first_name )
  3  (  PARTITION x1 VALUES LESS THAN ( 'H' )
  4       TABLESPACE data01
  5  ,  PARTITION x2 VALUES LESS THAN
  6                              ( MAXVALUE )
  7  ) ;
```


Global Index Example

The `EMP` table is a copy of `HR.EMPLOYEES`.

The partitioning syntax of the index is the same as you would use for partitioning a table, with the `GLOBAL` keyword.

Consider:

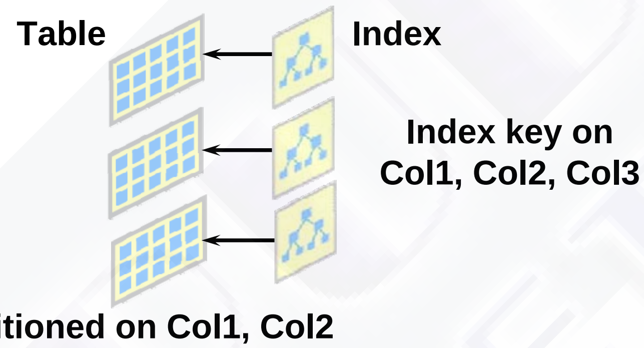
```
SELECT * FROM EMP WHERE FIRST_NAME='Lex' ;
```

This will only perform index lookup in partition `X2`, which points to the appropriate rows in the table. It does not matter if the table is partitioned or not, the index lookup gives the direct rowid of the table row, so the table partitioning will not alter the effectiveness of the index lookup.

Local Prefixed Index

Local prefixed indexes:

- **Only possible on partitioned tables**
- **Both the partition key of the index partitions and the leading columns of the index key are the same as the table partitioning key.**



Local Prefixed Indexes

Local prefixed indexes can be specified against all four table partition types.

B*tree and bitmap indexes can be local prefix partitioned.

Local prefixed indexes require less space for the rowid reference, because the rows to which it refers reside in the corresponding table partition, which implies one tablespace.

Local Prefixed indexes can be unique or nonunique.

Usage Note

Local prefixed indexes are particularly useful with massive parallel operations.

Local Prefix Index Examples

```
SQL> CREATE INDEX idx ON hr_emp( first_name )
  2    LOCAL ;
```

```
SQL> CREATE INDEX idx ON hr_emp( first_name )
  2   TABLESPACE indx04
  3   LOCAL
  4   ( PARTITION ex1 TABLESPACE indx01
  5   , PARTITION ex2 TABLESPACE indx02
  6   , PARTITION ex3
  7   ) ;
```


Local Prefixed Index Examples

The table is created with:

```
CREATE TABLE hr_emp TABLESPACE data04
  PARTITION BY RANGE ( first_name )
  ( PARTITION e1 VALUES LESS THAN ( 'H' )
       TABLESPACE data01
  , PARTITION e2 VALUES LESS THAN ( 'Z' )
       TABLESPACE data02
  , PARTITION e3 VALUES LESS THAN ( MAXVALUE )
       TABLESPACE data03
  ) AS SELECT * FROM hr.employees ;
```

The first example places the index partitions in the same tablespace as the corresponding table partition.

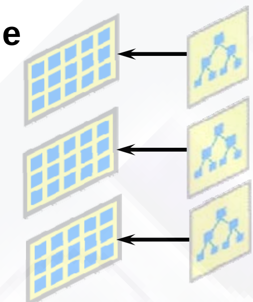
The second example shows that you can specify the physical attributes of the index partitions. The number of partitions specified must correspond to the number of partitions in the table. You cannot specify the key partition values. If the partition name is omitted, the index partition receives the same name as the corresponding table partition.

If the table is partitioned on multiple columns, then the index must use all the same columns in the same order before any additional index columns are specified. If this is not done, a nonprefixed local index will be created, and no error will be indicated.

Local Nonprefixed Index

Local nonprefixed indexes:

- **Are possible only on partitioned tables**
- **Although the index partition key is the same as the table partition key, the index key is not the same.**



Local Nonprefixed Indexes

The local nonprefixed index maintains equipartitioning with the table, but the index column can refer to all table partitions.

Local nonprefixed indexes can be specified against all four table partition types.

B*tree and bitmap indexes can be local nonprefix-partitioned.

Local nonprefixed indexes can be nonunique. If the partition key is a subset of the index key, then the local nonprefixed index can be unique.

Usage Note

If a query involves the columns of the partition key, then table partition elimination can be used to make a limited table partition full scan. If the query involves the same columns as the index key, then all index partitions must be range scanned, because each partition can potentially contain rows of any table partition.

However, if the query involves columns of both the partition key and the index key, then only the index partitions corresponding to the partition key are range scanned for the index key, thus affecting both table and index partition elimination. Nonprefixed indexes should therefore be chosen where two otherwise unrelated columns are often queried.

Local Nonprefix Index Example

```
SQL> CREATE INDEX idx ON hr_emp( last_name )
  2    LOCAL ;
```

3-12

Local Nonprefixed Index Example

The table is the same from the previous example; it is range partitioned on `first_name`.

The example will place the index partitions in the same tablespace as the corresponding table partition.

Note that there is no syntactical difference to specifying a prefixed or nonprefixed local index. Only when the local index uses the same leading columns as the table partition key, will the index be local-prefixed.

The same options, specifying the partition names and storage attributes for the index partition, as those used for local prefixed indexes, are available for local nonprefixed indexes.

Index Partitioning and Type Matrix

Index types	Global (Range)	Local (all)
B*Tree	Yes	Yes
Bitmap	No	Yes
Bitmap Join	No	Yes
Secondary IOT	No	Yes
Cluster*	No	No

ORACLE

Index Partitioning and Type Matrix

Cluster index is a simple B*tree index used to implement clustered tables. Clustered tables can not be partitioned.

Specifying Index with Table Creation

The partition structure of an index that is used for primary key constraint can be specified together with the partitioned table creation.

```
SQL> CREATE TABLE nonsimple
  ( idx number, txt varchar2(10),
    CONSTRAINT s_pk PRIMARY KEY ( idx ) )
  TABLESPACE data04 PARTITION BY HASH ( txt )
  ( PARTITION s1, PARTITION s2 )
ENABLE CONSTRAINT s_pk USING INDEX
 GLOBAL PARTITION BY RANGE ( idx )
 ( PARTITION spk1 VALUES LESS THAN ( 0 )
       TABLESPACE indx02 ,
  PARTITION spk2 VALUES LESS THAN (MAXVALUE)
       TABLESPACE indx03 ) ;
```

ORACLE

Specifying Index with Table Creation

When the index is created together with the table, the syntax structures allow for the specification of both the index partitioning and the table partitioning; for example, to enforce a primary key.

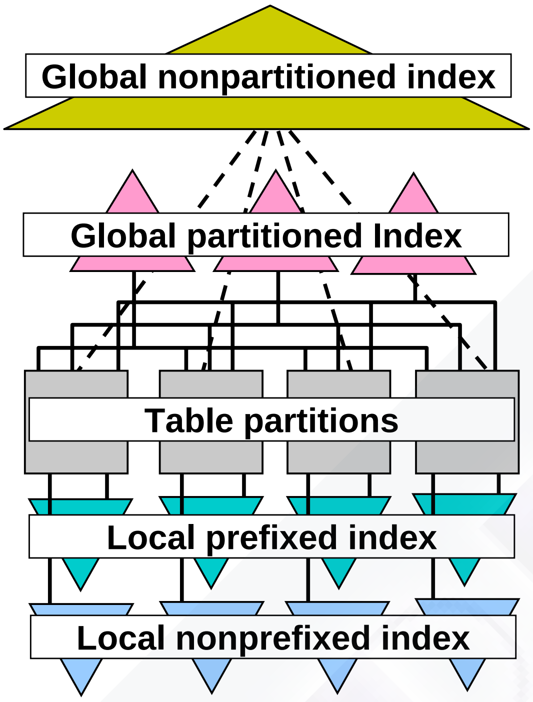
The option of defining the index attributes of a constraint are the same for nonpartitioned tables or indexes; for the partitioned index, the partitioning clauses just extend the storage clause that would have been used.

Note that this structure is applicable to a partitioned table with a nonpartitioned index and a nonpartitioned table with a partitioned index, too.

Example

The table is hash partitioned on the `txt` column, and has a global prefixed range partitioned primary key on the `idx` column.

Graphic Comparison of Partitioned Index Types



- Global nonpartitioned index
- Global partitioned prefixed index
- Local prefixed index
- Local nonprefixed index

Graphic Comparison of Partitioned Index Types

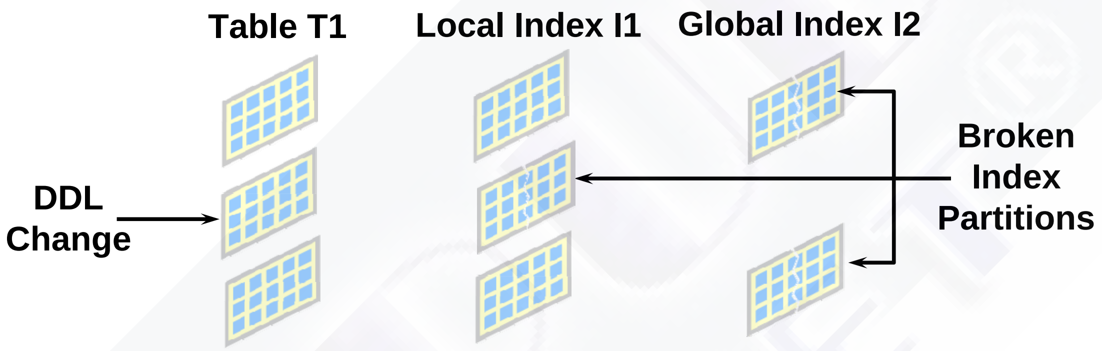
The syntax to create the global nonpartitioned index, that is, an ordinary index, is unchanged from creating a simple index on a simple table.

If the table is not partitioned, then only the global index types can be created.

Index Partition Status

A table partition can be altered:

- **With DML - The index is maintained.**
- **With DDL - The index might become `UNUSABLE`.**
  - **Usually only one partition for local indexes**
  - **The whole index for global or nonpartitioned indexes**



Index Partition Status

Partitioned table and index maintenance will be covered in the lesson on *Maintenance of Partitions*.

A table partition can be modified with ordinary DML (Insert, Update, Delete). The index will be updated accordingly.

A table partition can be altered with DDL (for example `MOVE`), or a direct DML, such as a direct parallel load operation, which leaves part of the index in a questionable state. The index is not updated and is marked `UNUSABLE`. This is different from `INVALID`.

- If the index is local, then only the corresponding index partition is affected.
- If the index is global or non partitioned, the whole index is affected. That is, for global indexes, all index partitions are marked `UNUSABLE`.

One of the management advantages of partitioning is that only a section of the data is affected. The maintenance operation to remedy faults is also limited to the involved partitions.

Index Partition UNUSABLE

- The index remains defined.
- If partitioned, other partitions remain fully usable.
- The index will block DML on the corresponding table.
- Queries can fail or bypass UNUSABLE index partitions depending on the session SKIP_UNUSABLE_INDEX setting.
  - TRUE, use another execution plan
  - FALSE, report ORA-1502

UNUSABLE Index state

The index still occupies space.

Rebuild can be limited to the partitions affected. Nonpartitioned indexes must be dropped and rebuilt.

Using the `UPDATE GLOBAL INDEXES` clause on the DDL command will automatically maintain all indexes.

SKIP_UNUSABLE_INDEX limitations

Only queries can bypass a bad index partition. `INSERT`, `UPDATE`, and `DELETE`, which require the affected index partition, will always give an error until the fault is remedied.

Use the `ALTER SESSION SET SKIP_UNUSABLE_INDEX={TRUE | FALSE}` to set the session mode. Default is `FALSE`.

Data Dictionary Views Indexes

Name	Purpose	N
DBA_INDEXES	Index structure, Partition Y/N	I
DBA_PART_INDEXES	Partition type, default values	I
DBA_IND_*PARTITIONS	Partitions detail	P
DBA_*PART_KEY_COLUMNS	Partition keys	P
DBA_IND_COLUMNS	Index keys	I

* SUB variation

I = per index
P = per partition

ORACLE

Data Dictionary Views Indexes

The data dictionary views for partitioned indexes follow the same pattern as for tables.

The basic index attributes in DBA_INDEXES contain the index type (unique in UNIQUE, bitmap or normal in INDEX_TYPE) and if the index is partitioned (yes/no in PARTITIONED). There is one row for every index.

The index key description is stored in DBA_IND_COLUMNS as it is for nonpartitioned indexes.

The partition definition is in DBA_PART_INDEXES that describe the partition type (range, hash, list, and so on) and default storage attributes of partitions. (The corresponding fields in DBA_INDEXES are NULL.) There is one row for each index.

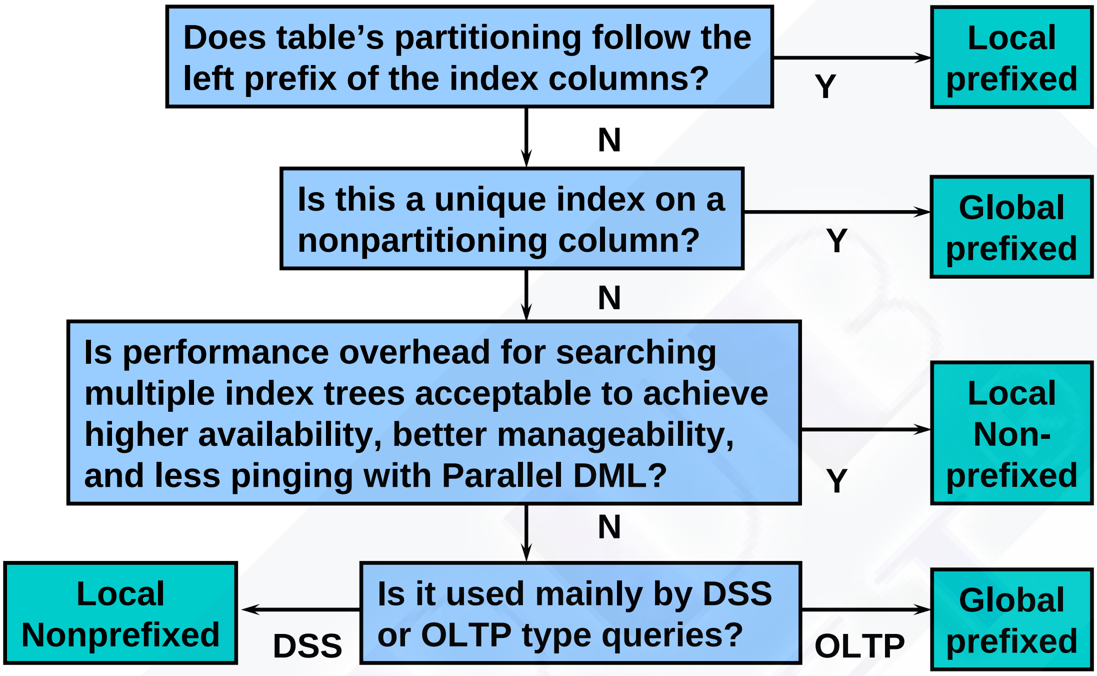
The individual partitions are described in DBA_IND_PARTITIONS, which describe the end point (range) or values of the partition, the storage attributes, and the index partition STATUS. There is one row for each partition. The subpartitions are described in DBA_IND_SUBPARTITIONS.

The partition keys are described in DBA_PART_KEY_COLUMNS and DBA_SUBPART_KEY_COLUMNS, as they are for table partition keys. The OBJECT_TYPE columns show if the partition key is for a table or an index.

Segments, Dictionary Objects

There is no difference from the data dictionary views used for partitioned tables.

Guidelines for Partitioning Indexes



Guidelines for Partitioning Indexes

When you are deciding how to partition indexes on a table, consider the mix of applications that must access the table. There is a trade-off between performance on the one hand, and availability and manageability on the other.

Some guidelines for you to consider are described in the following section.

Online Transaction Processing (OLTP)

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 maintenance operations on the table. Local nonprefixed indexes are very useful for historical databases.

Guidelines for Partitioning Indexes (continued)

Decision Support Systems (DSS)

Local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

If possible, indexes for historical tables should be local. 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 keys do not contain the partitioning key, are not supported.

Summary

In this lesson, you should have learned how to:

- Describe the different index partitioning types
- Create partitioned indexes

Practice Overview: Creating Partitioned Indexes

This practice covers the following topics:

- Creating most types of partitioned index
- Using the data dictionary to verify the partition structure
- Examining failures of some partition attempts
- Specifying partitioned constraints

ORACLE

4

Maintenance of Partitioned Tables and Indexes

Objectives

After completing this lesson, you should be able to do the following:

- List all of the alterable partitioned table and index attributes
- Describe the overhead associated with each maintenance command

4-2

Maintenance Overview

With the `ALTER TABLE` or `ALTER INDEX` statements, you can modify:

- Logical attributes of table - Column data types
- A single partition:
  – Allocate, truncate, rename
  – Exchange with a table
- Table or index partitions:
  – Add, drop, move, split, coalesce, merge
  – Row movement

You cannot simply alter the partition type.

ORACLE

Maintenance Overview

Nearly every attribute of a partitioned table or index can be altered after the table or index has been created and populated. This is different from altering the table or index as a whole, as you do with a nonpartitioned table or index.

Changes to a table's logical properties, such as the number and types of data columns, can be made to partitioned as well as nonpartitioned tables with the same syntax.

The partition's logical property, for example, the name, can be altered.

The partitions physical properties such as the storage attribute can be altered. For some attributes, the partition must be moved for the changes to take effect.

The table or index's partition key definition can be altered by adding, dropping, merging, or splitting partitions of the table or index. This may affect the data within some existing partitions.

Individual partitions can be exchanged, moved, truncated, or dropped, affecting the data within the partition.

You cannot change the partitioning type with a simple DDL statement; you must use the `DBMS_REDFENITION` package to achieve that. The `DBMS_REDFENITION` can transparently run online, that is, while users access and modify the table rows.

Table and Index Interaction During Partition Maintenance

- **Altering a table partition will affect indexes on the table:**
  - **Local indexes are added, dropped, or set to `UNUSABLE`.**
  - **Global indexes are marked `UNUSABLE`.**
  - **Global partitioned indexes are marked `UNUSABLE`.**
- **Adding the `UPDATE GLOBAL INDEXES` clause will maintain global indexes.**
- **Altering an index partition does not affect other indexes or tables.**

Table or Index Interaction During Partition Maintenance

Depending on the operation made on a table partition, the indexes on the table will be affected.

`UPDATE GLOBAL INDEXES`

When altering a table partition, you can add the

```
UPDATE GLOBAL INDEXES
```

clause, which will automatically maintain affected global indexes and partitions.

The clause is available in Oracle9*i* and later.

Modifying a Table or Indexing Logical Properties

- You can modify the name of a partitioned table or index, just like you can modify a nonpartitioned one.
- You can add, modify, or drop columns in a table, like a nonpartitioned one.
- There are restrictions on modifying the columns used for the partition key or if not all partitions are available.

```
SQL> RENAME name TO newname
```

```
SQL> ALTER TABLE …
  2   ADD ( column type … )
```

ORACLE

Modifying Table Logical Properties

You use the ALTER TABLE statement to modify attributes of a table that are independent of the physical organization of the table. For example, you can add a new column or constraint, change the data type of a column, or enable an existing constraint. If the table is partitioned, these attributes are common to all partitions.

Rules for altering the logical attributes of a table include the following:

- You cannot change the data type or length of a column that is used to partition the table or index.
- You cannot add a column or change a column to the LONG or LONG RAW data type.
- If one or more partitions reside in a read-only tablespace, then:
  - You cannot add a new column with user-specified default, such as ALTER TABLE tab ADD (col NUMBER DEFAULT 6).
  - You cannot modify an existing VARCHAR2 (or VARCHAR) column to be of type CHAR (or CHARACTER).
  - You cannot increase the length of an existing CHAR (or CHARACTER) column.

Modifying Partition Properties on the Table

- **The row migration property can be enabled and disabled.**

```
SQL> ALTER TABLE simple ENABLE ROW MOVEMENT ;
```

- **The default storage attributes of the table can be altered.**

```
SQL> ALTER TABLE simple MODIFY
  2    DEFAULT ATTRIBUTES PCTFREE 50 ;
```


Modifying Partition Properties on the Table

These statements effect table-level changes on a partitioned table.

Row Movement

```
SQL> ALTER TABLE simple DISABLE ROW MOVEMENT ;
```

will disable future row movement.

Default Partition Storage Attributes

These default values are used when new partitions are created on this table or index. All storage attributes can be specified.

Using the ALTER TABLE or INDEX Commands

For RENAME, TRUNCATE, ADD, DROP, SPLIT, COALESCE, MERGE, and MOVE commands, use the following:

```
SQL> ALTER TABLE table_name
  2    operation PARTITION partition_name …

SQL> ALTER INDEX index_name
  2    operation PARTITION partition_name …
```

```
SQL> ALTER TABLE table PARTITION ( name ) …

ORA-14052: partition-extended table name
syntax is disallowed in this context
```


ALTER TABLE / INDEX

The alter table / index command has special syntax for altering the partition. The command fragment above shows the general syntax for both table and index alterations. The rest of the statement is dependent on the operation. The operation can be `RENAME`, `DROP`, `ADD`, `SPLIT`, `COALESCE`, `MERGE`, and `MOVE`.

You do not use the `tablename PARTITION ( partition_name )` partition-extended table name syntax, that is used in `SELECT`s, as shown in the second statement fragment.

Local and Global Index

You can directly add or remove partitions of global indexes, but not on local indexes. Local index partitions automatically follow these partition operations on the table.

You can rename, move, and alter storage attributes on both global and local index partitions.

Parallelization

Operations that modify rows or the global index update can be made parallel by adding the `PARALLEL n` clause.

If the table has a default parallelization clause, it can be suppressed by using the `NOPARALLEL` clause.

Renaming a Partition

```
SQL> ALTER TABLE tab_hash
  2    RENAME PARTITION SYS_P451 TO HASH_1 ;
```


Renaming a Partition

There are no restrictions on renaming partition names. The partition name must be unique within the affected table or index.

To rename the table or index, whether partitioned or not, use one of these two statements:

```
RENAME old table TO new table ;
ALTER TABLE old table RENAME TO new table ;
```

Partition Storage Changes

- **TRUNCATE (DROP)**

```
SQL> ALTER TABLE tab TRUNCATE PARTITION px ;
```

- **MODIFY: Partition storage**

```
SQL> ALTER TABLE tab MODIFY PARTITION px
  2   ALLOCATE EXTENT (SIZE 100M) ;
```


Partition Storage Changes

Partition or subpartition segments that occupy space in the tablespaces can have their storage attributes modified like nonpartitioned tables and indexes. The same restrictions apply: they cannot truncate smaller than the initial allocation, cannot deallocate less than the highest one used, and `PCTFREE` takes effect only for new blocks, and so on.

TRUNCATE

This command discards the data rows in the partition, and drops all but the initial storage, unless the `REUSE` option is used. Global indexes are marked UNUSABLE unless the `UPDATE GLOBAL INDEXES` clause is added. Corresponding local indexes are also truncated, and remain or become valid.

`TRUNCATE TABLE tablename` will truncate all partitions.

MODIFY - Partition Storage

Storage attributes include `ALLOCATE EXTENT`, `DEALLOCATE UNUSED`, `PCTFREE`, `PCTUSED`, (NO)`LOGGING`, `STORAGE ( … )`, LOB storage clauses, IOT `OVERFLOW` storage, and so on.

Note: Storage attributes are changed when moving a partition. The `MODIFY PARTITION` clause is also used for operations not involving partition storage, as will be shown later.

Moving a Partition

- **Moving a partition places it in a new tablespace.**
  - **All storage attributes can be modified.**
  - **The partition is reorganized.**
- **All partition types can be moved: range, list, hash, and subpartitions**
- **Both table and index partitions can be moved:**
  - **Use `MOVE` for table partitions**
  - **Use `REBUILD` for index partitions**

ORACLE

Moving a Partition

You can move a nonpartitioned table, using:

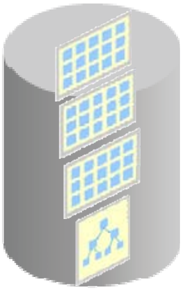
```
ALTER TABLE tablename MOVE [ ONLINE ];
```

In order to move the whole partitioned table, move all of its partitions.

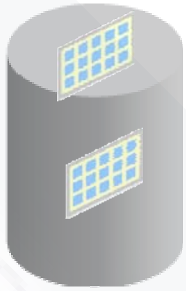
You can only move one partition at a time. Parallel sessions can move each partition separately.

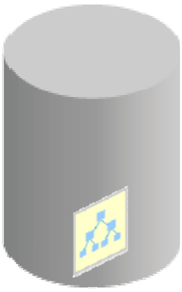
Global indexes are marked UNUSABLE, unless the `UPDATE GLOBAL INDEXES` command is specified. Local indexes are moved and marked UNUSABLE, unless the partition is empty.

Moving a Partition: Example

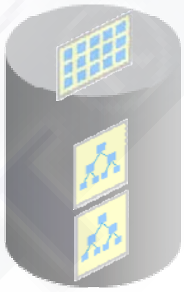


```
SQL> ALTER TABLE simple
  2   MOVE PARTITION p2
  3    TABLESPACE data03
  4    PCTFREE 95 ;
```





```
SQL> ALTER INDEX s_glo
  2   REBUILD PARTITION sg_1
  3    TABLESPACE data03 ;
```



ORACLE

Moving a Partition: Example

To move a subpartition of a composite partitioned table or index, the keyword `SUBPARTITION` is used instead of `PARTITION`.

Indexes and tables are not moved, they are rebuilt.

When moving a table partition or rebuilding an index partition, all storage attributes can be specified, thus altering them during the move or rebuilding.

Adding a Partition

- For Range Partition, a new partition is added *at the end*.
  - Specify a new high end value
  - Cannot add if `MAXVALUE` partition exists
  - Does not mark global indexes UNUSABLE
- For Hash Partition and Hash Subpartition, a added partition will receive rows redistributed from other partitions.
- For List Partition, a added partition is added as specified.
- Cannot add a partition to a global index
  - Local indexes follow the table

4-12

Adding a Partition

Adding a partition has different side effects for the different partition types.

Range and List

For Range and List partitions, an empty partition segment is created because it cannot contain rows (such rows would have been illegal to `INSERT` before, being outside allowed partition keys). The addition has no effect on global indexes. Local index partitions are created to match the table partition.

Hash and Hash Subpartition

For the Hash or composite hash subpartition, the addition of another hash partition means that existing rows of another existing hash partition would have been placed in this new partition had it existed before. The hash distribution changes by the addition of a new hash partition or subpartition. Consequently, these rows are immediately migrated. This will mark global and local index partitions UNUSABLE.

Local Index Storage

Added local index partitions are stored in the same tablespace as the table partition, unless the index has a storage default defined at the index level.

When to Add a Partition

A partition is added when:

- Changes in data require it:
  - Rolling window (range partition)
  - New key values (list partition)
- The quantity of data increases
  Spread over more storage (hash partition)

4-13

When to Add a Partition

For tables with rolling windows, you need to add a partition with the new time interval as time passes.

If you have the list partitioned on office locations, you can add another partition when the company expands.

You can add new hash partitions in tablespaces on new disk drives to further spread the IO load. A similar effect can be achieved by adding more data files from different drives to the tablespaces containing the table or table partitions, but this will leave tables large and unwieldy.

If you expand the number of CPUs on your server hardware, you want to change the degree of parallelism. For maximum efficiency, you may want to increase the number of partitions, especially hash partitions, in a table to match.

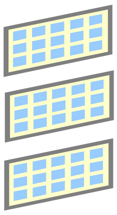
Not Adding a Partition

You cannot add a range partition *in the middle*; to do that, you must *split* a partition.You cannot add another range partition if the `MAXVALUE` partition exists; you must *split* the last partition instead.

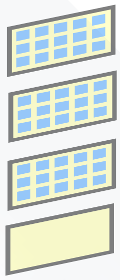
You cannot add another partition if you reach the maximum of 65534 partitions of a single partitioned table. A few thousand partitions might be a practical maximum.

Adding a Partition: Examples

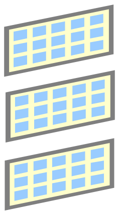
- **List-partitioned table:**



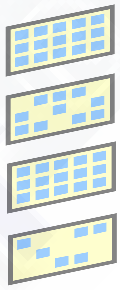
```
SQL> ALTER TABLE tab_list ADD
  2     PARTITION p3
  3     VALUES ( 'NEW' )
  4     TABLESPACE data03 ;
```



- **Hash-partitioned table:**



```
SQL> ALTER TABLE tab_hash ADD
  2     PARTITION p3
  3     TABLESPACE data03
  4   UPDATE GLOBAL INDEXES ;
```



Adding Partition Examples

When adding a partition, all partition attributes, such as partition name and storage attributes, can be specified or omitted, in which case the table level defaults will be used.

Only one partition can be added. Multiple additions require multiple statements.

List and Range

The addition of a range partition is very similar to the addition of a list partition. The difference is that the `VALUES LESS THAN ( … )` clause is used instead of `VALUES ( … )`. `MAXVALUE` can be specified.

For a list partitioned table, the added key values must be unique to the existing key values, including the `NULL` key value.

Global indexes are not affected. Local indexes have a corresponding partition added, using the default storage parameters defined on the index.

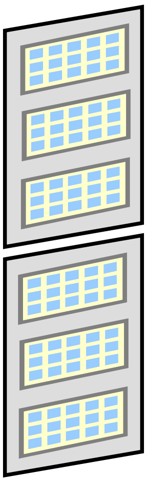
Hash and Subpartition

Note the rearrangement of the existing rows in the diagram.

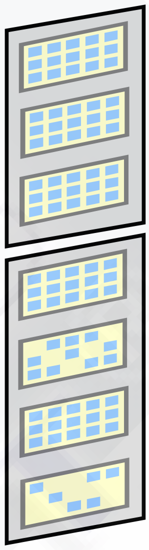
Local indexes will have a corresponding partition added. Affected local index partitions and global indexes are marked UNUSABLE.

Adding a Subpartition: Example

Composite partitioned table:



```
SQL> ALTER TABLE simple
  2     MODIFY PARTITION s1
  3     ADD SUBPARTITION
  4                    s1_h3 ;
```



ORACLE

4-15

Adding a Subpartition Example

If you need to place the local index segment, you can move it after creation or alter the index default storage before creation.

Dropping a Partition

- **Discards the rows contained quickly, without rollback**
- **Only Range and List Partitions can be dropped.**
- **One partition must remain.**
- **You can drop a partition from a global index.**
  - **Cannot drop the last partition**
  - **The previous partition is marked UNUSABLE, unless the dropped partition is empty**
- **Local indexes partitions follow the table partitions.**

4-16

Dropping a Partition

Dropping a partition will discard the rows stored in that partition as a DDL statement. It can not be rolled back. It executes quickly, and uses few system resources (Undo and Redo).

You must be the owner of the table or have the `DROP ANY TABLE` privilege to drop a partition.

You cannot drop a partition of a hash-partitioned table.

If a table contains only one partition, you cannot drop the partition. You must drop the table.

For range partitioned tables, dropping a partition does not make inserts of the dropped range invalid; they are now part of the next higher partition. If the dropped partition was the highest partition, possibly even if it had `MAXVALUE` as its end range, then inserts to the missing partition do fail.

Indexes

You cannot drop local indexes directly. Corresponding local index partitions are dropped regardless of status, when table partition is dropped.

You can drop a partition of a global index. The dropped index entries are recreated in the next higher partition on rebuilding.

When to Drop a Partition

When changes in data require it:

- Rolling window (range partition)
- Obsolete key values (list partition)

ORACLE

4-17

When to Drop a Partition

For tables with rolling timeframes, you need to drop a partition with the old data as time passes.

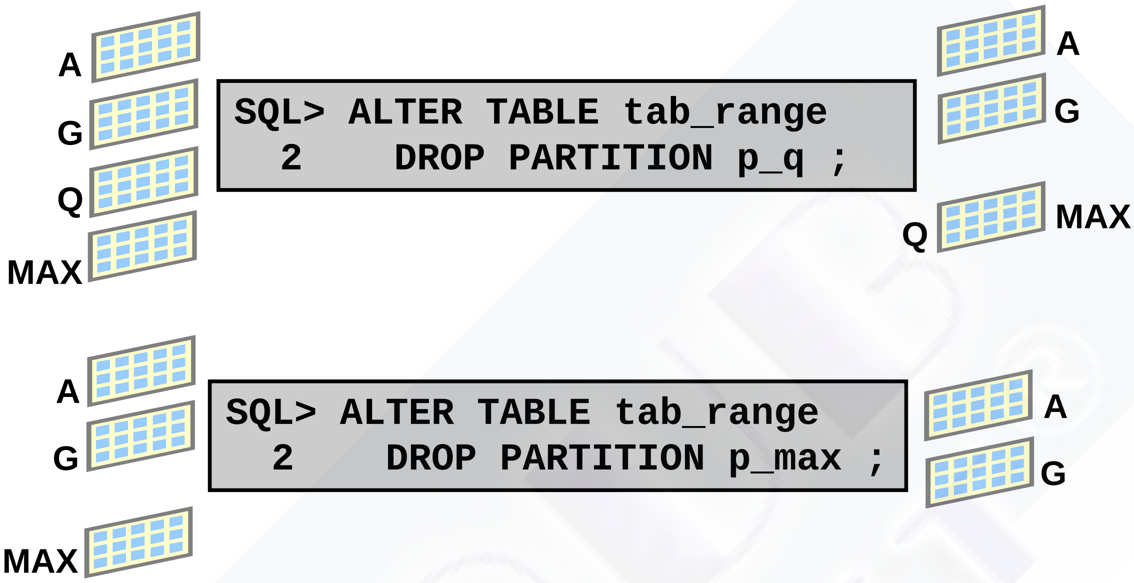
For a list partitioned table, you can drop a partition when some partition key values are of no further use.

When dropping or adding a hash partition, it is recommended that you work toward ending up with a power of two number of hash partitions, for optimal data spread across partitions.

Not Dropping a Partition

If you want to remove the range key but want to keep the data, that is, have all the data in fewer partitions, then you should *merge* the partition.

Dropping a Partition: Examples




```
SQL> ALTER TABLE tab_range
  2     DROP PARTITION p_q ;
```

```
SQL> ALTER TABLE tab_range
  2     DROP PARTITION p_max ;
```


Dropping a Partition: Example

Only one partition can be dropped. Multiple drops require multiple statements.

“G” to “P” rows that were stored in the third partition are discarded. If any new “G” to “P” rows are inserted they will be stored in the partition that is now the third partition.

After the second drop partition statement, only rows less than “G” can be added.

Splitting and Merging a Partition

- Splitting a partition creates two new partitions filled with rows of the split partition, which is discarded.
- Merging a partition collects the rows from two partitions and drops one of them.
- For range partitions, it involves two consecutive partitions.
- Hash partitions or subpartitions cannot be split or merged.
- You can split a partition on a global index.

ORACLE

4-19

Splitting and Merging a Partition

You can split any partition. For a range partitioned table, adding a partition instead of splitting the highest partition gives different results. Adding gives a new highest partition. If the last partition has a `MAXVALUE` value, then you can split it to *add* another partition under the `MAXVALUE` partition.

A global range partitioned index has the last partition set to `MAXVALUE`, thus you cannot add partitions, but you can split partitions.

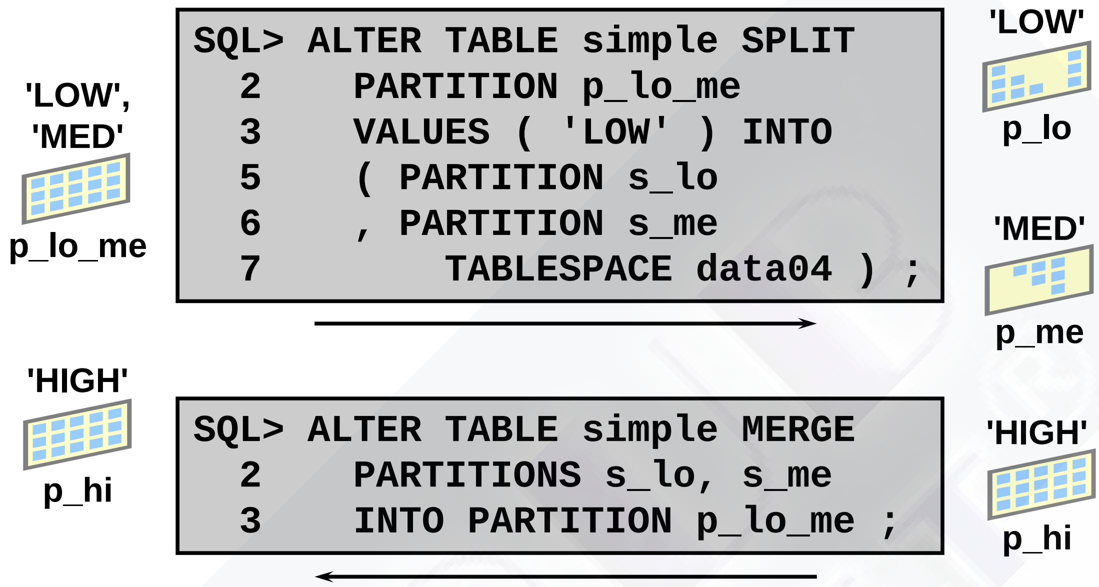
When you split a list partitioned table, you specify the key values of one split partition. The remaining keys go into the other split partition.

When you split a range partitioned table, specify the split value. This becomes the `VALUES LESS THAN` value of one of the new split partitions, and the other inherits the `VALUES LESS THAN` value of the original partition.

If the table is composite partitioned, you can specify the subpartitioning attributes for the new split partitions. Default is the same subpartitioning as the original split partition.

One or both local index partitions that result from the split will be marked UNUSABLE depending on whether the corresponding table partitions have any rows in them after the split. Global indexes are marked UNUSABLE.

Splitting and Merging: List Partitions




```
SQL> ALTER TABLE simple SPLIT
  2     PARTITION p_lo_me
  3     VALUES ( 'LOW' ) INTO
  5     ( PARTITION s_lo
  6     , PARTITION s_me
  7         TABLESPACE data04 ) ;
```

```
SQL> ALTER TABLE simple MERGE
  2     PARTITIONS s_lo, s_me
  3     INTO PARTITION p_lo_me ;
```


Splitting and Merging Examples on a List Partitioned Table

The table above is a list partitioned table, with the key values `'LOW'`, `'MED'` on `p_lo_me` and `'HIGH'` on `p_hi`.

Split (List and Range)

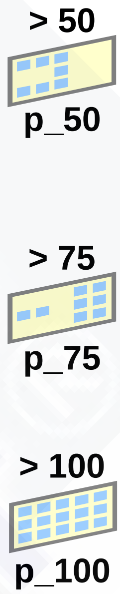
Omitted storage attributes are inherited from the original partition, not the table level defaults. If you omit the whole partition specification `( PARTITION s_lo, PARTITION s_me TABLESPACE data04 )`, the two split-off partitions get a default name, SYS_Pnnnn.

Both new split partitions are *new*, and all rows have been moved.

Splitting and Merging: Range Partitions


```
SQL> ALTER TABLE simple SPLIT
  2     PARTITION p_100
  3     AT ( 75 ) INTO
  5     ( PARTITION s_75
  6     , PARTITION s_100
  7         TABLESPACE data04 ) ;
```

```
SQL> ALTER TABLE simple MERGE
  2     PARTITIONS p_75, p_100
  3     INTO PARTITION p_100
  4     TABLESPACE data03 ;
```



Splitting and Merging Examples on a Range Partitioned Table

The table above is a range partitioned table, with the key values for `VALUES LESS THAN` are `50` on `p_50` and `100` on `p_100`.

Merge (List and Range)

You can specify all storage options for the new merged partition. Omitting them will use the table level defaults, not the inherited attributes from either of the original partitions.

Altering List Partition Key Values

The key list in a list partition can be altered, as long as no rows are affected.

```
SQL> ALTER TABLE simple MODIFY
  2   PARTITION p_high
  3   ADD VALUES ( 'ULTRA', 'EXTREME' ) ;
```

```
SQL> ALTER TABLE simple MODIFY
  2   PARTITION p_high
  3   DROP VALUES ( 'ULTRA' ) ;
```

ORACLE

4-22

Altering List Partition Key Values

Because altering list partition key values does not affect any rows, global and local indexes are not affected.

The `DROP VALUES` operation will fail if any rows match. This check will be faster if there is an index on the list partition keys.

Coalescing a Partition

- The `COALESCE` command for hash and subpartitions has the same effects as the `MERGE` command on non-hash partitions:
  - Rows are distributed to other partitions.
  - The partition is dropped.
- Using the `COALESCE` command for a partition of an IOT table will reorganize the IOT.

4-23

Coalescing a Partition

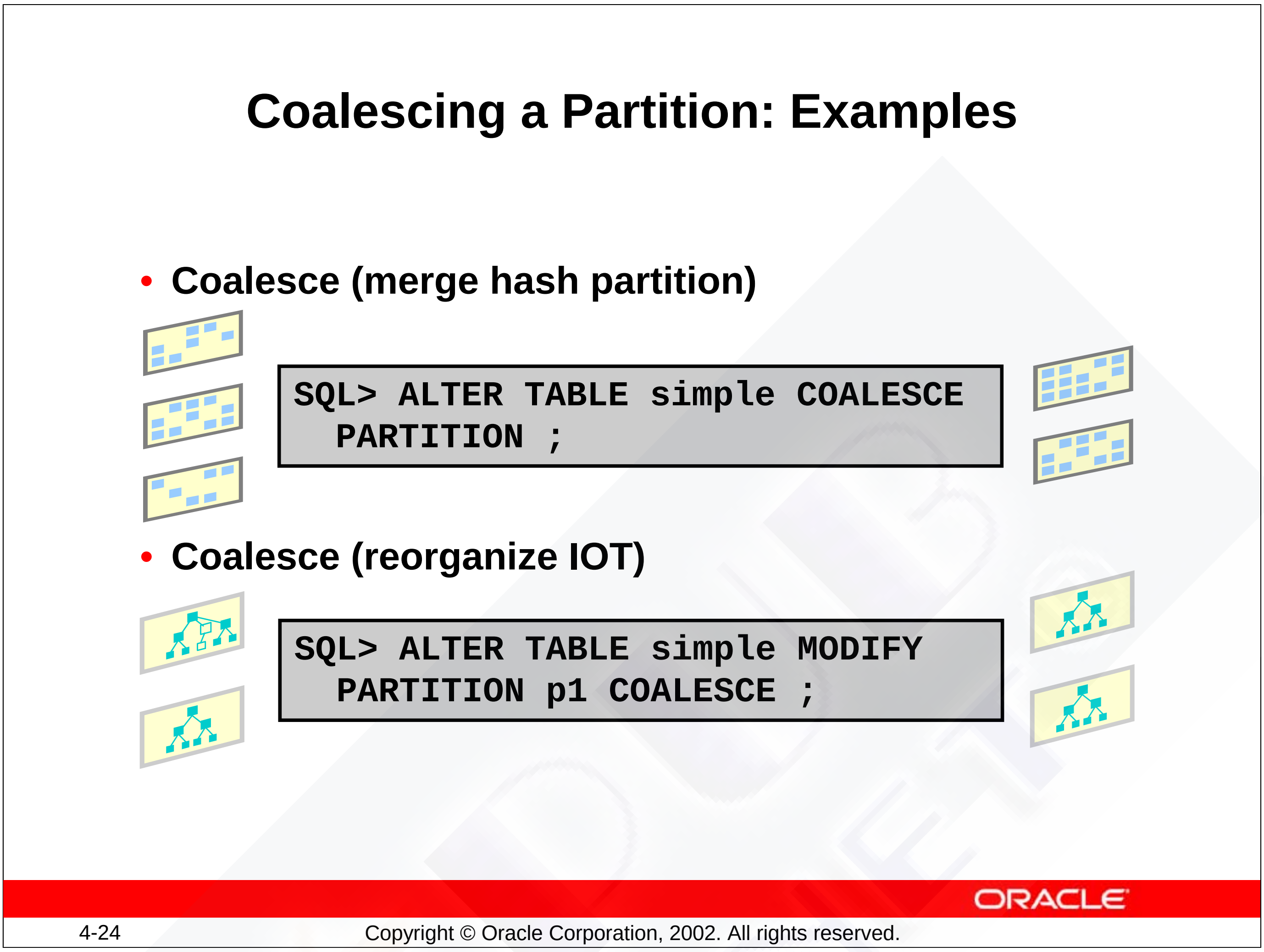
The two operations, merging a hash partition and reorganizing a partition of an IOT table, are quite separate and distinct, with different syntax.

Hash Partition and Subpartition

You cannot specify which partitions are involved. This is a limitation of the hashing system. The operation reduces the number of hash partitions by one.

Partition of an IOT

You can specify which partition is to be reorganized. You can also coalesce a nonpartitioned IOT table.



Coalescing a Partition: Examples

For coalescing a subpartition, the syntax is:

```
ALTER TABLE simple MODIFY
  PARTITION p1 COALESCE SUBPARTITION ;
```

This syntax is very similar to the coalescing of an IOT partition.

The coalescing of an IOT partition can specify storage attributes for the reorganized partition as it gets moved.

Exchanging a Partition with a Table

- A range or hash partition can be exchanged with a nonpartitioned table.
  - This is done by *swapping the names*.
  - You can work offline on the swapped data.
  - A hash subpartition can be swapped with a hash partition.
- The nonpartitioned table must have the same structure as the partitioned table.
- The exchange operation will verify partition key conformity by default.

ORACLE

Exchanging a Partition with a Table

The operation does not move the rows.

There is no restriction on the table or partition being empty or having any number of rows.

The two tables must have the same column names, in the same order,and with the same data type.

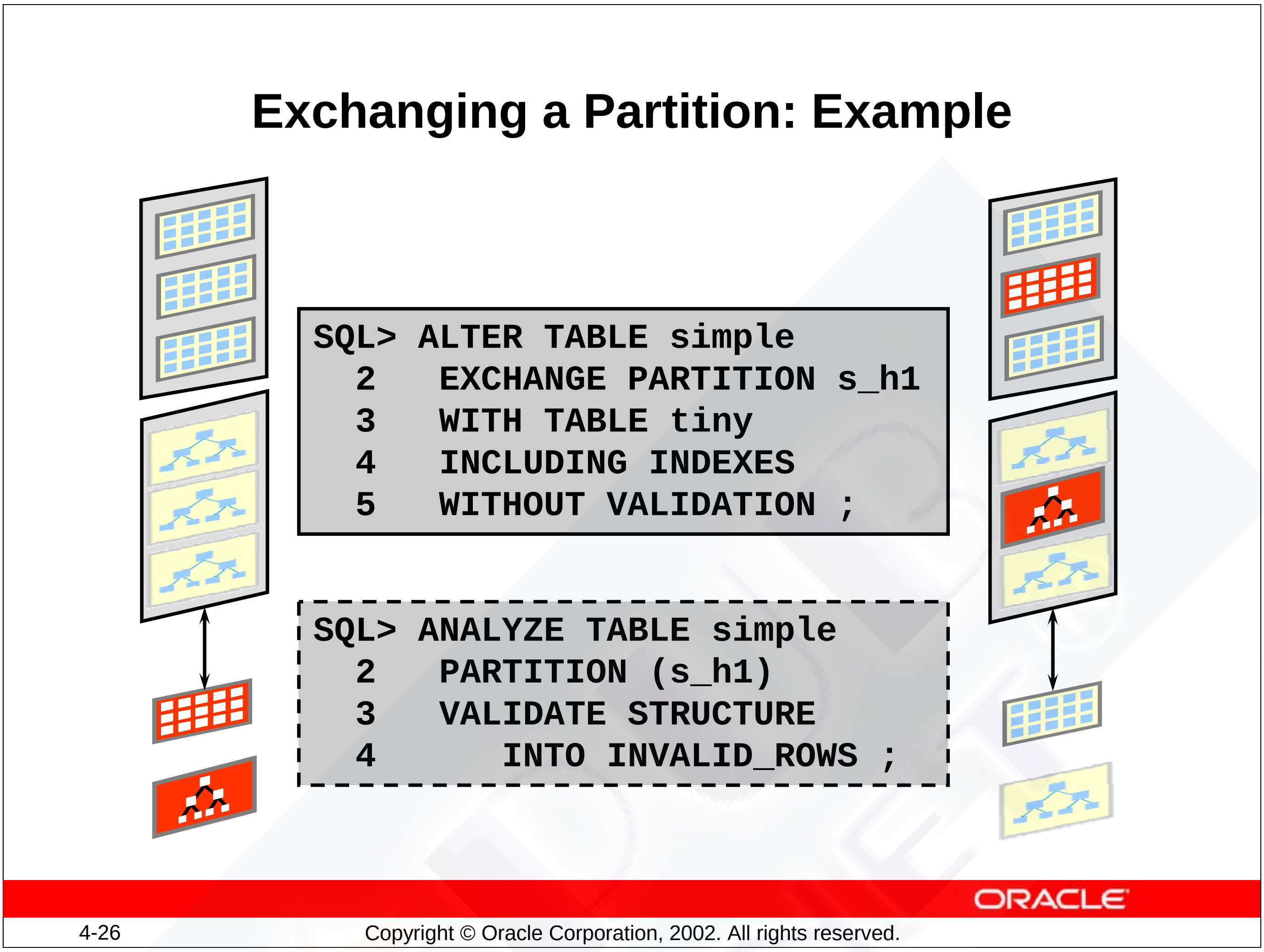
The two tables can have different indexes, grants, owners, triggers, and constraints. Typically, the nonpartitioned table has less of these.

Local indexes partitions are exchanged with matching nonpartitioned indexes defined on the nonpartitioned table.

Global indexes on the partitioned table are marked UNUSABLE. Indexes on the nonpartitioned table, which are not exchanged with a local index, are marked UNUSABLE and are not maintained with a `UPDATE GLOBAL INDEXES` clause.

Partition Key Conformity

The table rows may have been entered without any value constraints. When these are exchanged with a partition, the partition key column values must be valid values of the partition. This is verified before the exchange takes place by scanning the nonpartitioned table rows. Using the `NOVALIDATE` clause will skip this validation.



Exchanging a Partition: Example

In this example the indexes are exchanged, too. There is no validation of the rows conforming to partition key values.

You can create the standalone table with

```
CREATE TABLE tiny AS SELECT * FROM simple WHERE ROWNUM<1 ;
```

The partitioned table must only contain one partition.

The nonpartitioned table does not need to be owned by the same user as the partitioned table.

Validation After Exchange

In this example, a validation to detect rows that do not belong in the partition is executed after the exchange has completed. The ROWIDs of invalid rows are stored in the table INVALID_ROWS.

Scanning the table rows before or after the exchange takes time, depending on the size of the table.

If invalid partition key values are entered into the partitioned table with the exchange, then such records will not be returned by a query where partition elimination causes only scans in the proper partitions. This is true of all partition types.

Rebuilding Indexes

- If the partition operation has made an index unusable, it must be rebuilt.
- If the index is invalid, it must be dropped and re-created.
- Partitioned indexes must have each affected partition processed separately. You cannot rebuild a partitioned index as one whole index.

Rebuilding Indexes

Global indexes maintained with `UPDATE GLOBAL INDEXES` do not become `UNUSABLE`.

Rebuilding an Index: Examples

```
SQL> ALTER INDEX s_glo
  2    REBUILD PARTITION s_g1 ;
```

```
SQL> ALTER TABLE simple
  2    MODIFY PARTITION s_h1
  3    REBUILD UNUSABLE LOCAL INDEXES ;
```

```
SQL> ALTER INDEX s_cmp_idx
  2    REBUILD SUBPARTITION sys_subp453 ;
```

```
SQL> ALTER TABLE simple
  2    MODIFY SUBPARTITION sys_subp453
  3    REBUILD UNUSABLE LOCAL INDEXES ;
```

ORACLE

4-28

Rebuilding an Index: Examples

Rebuilding leaves the index in the same physical location only if it was unusable.

The command completes without error if the index was normal and usable before.

The first command will work on both global index partitions and local index partitions.

The first example rebuilds one global or local index partition, and the second example rebuilds all the local index partitions that correspond to the table partition.

The third and forth example are the corresponding syntax for index subpartitions.

Benefits and Costs of `UPDATE GLOBAL INDEXES`

When using the UPDATE GLOBAL INDEXES clause:

- \+ Global indexes remain useable and available, even during the partition operation.
- \+ You do not have to perform a number of rebuild operations.
- − The partition operation will take longer.
- − The resultant global index may be larger.
- − You can not specify NOLOGGING.

Benefits of `UPDATE GLOBAL INDEXES`

The global index is updated in conjunction with the base table operation. You are not required to later and independently rebuild the global index.

There is higher availability for global indexes, since they do not get marked UNUSABLE. The index remains available even while the partition DDL is executing and it can be used to access other partitions in the table.

You avoid having to look up the names of all UNUSABLE global indexes partitions used for rebuilding them.

Costs of using `UPDATE GLOBAL INDEXES`

The partition DDL statement takes longer to execute since indexes which were previously marked UNUSABLE are updated. 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.

Updates to the index are logged, and redo and undo records are generated. If the entire index is being rebuilt, it can optionally be done NOLOGGING.

Rebuilding the entire index creates a more efficient index, since it is more compact with space better utilized. Further rebuilding the index allows you change storage options.

IOT Overflow and LOB Segments

- When altering table partitions:
  - any LOB partitions will correspondingly change.
  - Any `OVERFLOW` partitions will correspondingly change.
  - Storage attributes of LOB or `OVERFLOW` segments can be explicitly specified.

IOT `OVERFLOW` and LOB Segments

Specification for LOB segments and `OVERFLOW` segments attributes and storage attributes can be placed where partition storage attributes are specified. The same syntax is used as that used when these attributes were specified at the time the table was created.

Summary

In this lesson, you should have learned how to:

- Modify attributes of a partitioned table or index
- Drop, add, split, merge, coalesce, move, exchange, and truncate partitions on tables
- Drop, split, merge, and rebuild partitions on indexes
- List the index invalidations that occur with separate table partition operations

ORACLE

4-31

Practice Overview: Altering Table and Index Partition Attributes

This practice covers the following topics:

- Splitting and merging a partitioned table, including impossible attempts
- Splitting and merging a partitioned table, checking and fixing index usability changes
- Performing simple exchange operations

ORACLE

5

Partitioning Interaction

Objectives

After completing this lesson, you should be able to do the following:

- Describe the behavior of partitioned tables and indexes with other database features and utilities
- Describe Oracle Enterprise Manager support of partitioned objects

Using Partitioned Tables

Simply refer to the table as usual

- Partitioning pruning is automatic.
- Partition-wise joins are automatic.
- Share locks can occur on table, partition, or row level.

ORACLE

5-3

Using Partitioned Tables

Applications should be unaware that the table is partitioned. Heap tables return the rows *grouped by* the partition when compared to nonpartitioned tables.

The partitioning pruning and other optimizer access changes occur irrespective of whether table has been analyzed or not. The optimizer uses the structural definition of the partitions, that is, the partition key values, to determine if partitions can be skipped. Analyzing the partition tables and indexes is still recommended for the optimizer to choose between table scans and index scans, for example.

Partitions that are skipped can be offline or unusable for other reasons, without affecting the statement execution.

An operation that only needs a partition, will only place locks on that partition, not on the whole table.

Pruning Rules

Partition Pruning varies slightly with type of partition and query

- **Range Partition will select one contiguous range of partitions**
  - **Equality and Range**
- **List Partition can select a number of partitions**
  - **Equality, range and IN lists**
- **Hash Partition will only prune on equality**
  - **Equality and IN lists**
- **The pruning also works with joins**
- **The pruning also works with bind variables**

5-4

Pruning Rules

All partition pruning, or elimination, occurs based on the data dictionary static definition of the partitions.

Composite Partitioning works like Range and Hash respectively at each partition level, that is, the query should specify something on both partition and subpartition key.

The join condition has to yield something similar to a simple query for the pruning to work with the join, that is the join must be on the partition key.

Partition-wise Joins

- **Partition-wise join can occur when one or both tables of a join are partitioned on the join key**

5-5

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 Cluster 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. The Oracle server decides which type of join to use. This depends on the table partition keys, the join key, and if the tables are equipartitioned.

The subject is covered further in the *Oracle9i: Data Warehouse Administration* course.

ANALYZE and Partitioned Objects

- You can collect table, index, and column statistics for a single partition or subpartition of a table or index.

```
ANALYZE TABLE sales SUBPARTITION (jan02)
COMPUTE STATISTICS;
```

- You can validate the structure for a single partition or subpartition of a table or index.
- You can list chained rows for a single partition or subpartition.
- You can collect histograms for a single partition or subpartition.

5-6

Analyzing Partitioned Objects

The target of the ANALYZE statement can be a single subpartition, a single partition, or the whole table or index.

- Specifying the whole table or index, when the table or index is partitioned, or a partition that is subpartitioned, is interpreted as a request to analyze all partitions or subpartitions.
- The ANALYZE statement generates *combined* table, index, and column statistics for a combined partitioned table or index by merging the statistics from the subpartitions. The ANALYZE statement does not generate combined histograms.
- If the optimizer finds that there are subpartitions of interest that have not been analyzed, it uses the table and index defaults for each subpartition.
- The VALIDATE STRUCTURE INTO INVALID_ROWS statement verifies that the row belongs to the correct partition. If the row does not collate correctly, the rowid is inserted in the INVALID_ROWS table.

Analyzing Partitioned Objects (continued)

The UTLVALID.SQL script, located in $ORACLE_HOME/rdbms/admin, creates the following INVALID_ROWS default table:

```
create table INVALID_ROWS
 (owner_name         varchar2(30),
  table_name         varchar2(30),
  partition_name     varchar2(30),
  subpartition_name  varchar2(30),
  head_rowid         rowid,
  analyze_timestamp  date
 );
```

Data Dictionary Views Statistics

Name	Purpose
DBA_*PART_HISTOGRAMS	Histogram end points
DBA_*PART_COL_STATISTICS	Column statistics and histogram information

*** SUB variation**

5-8

Data Dictionary Views Analyze

The partition statistic correspond to the statistics for nonpartitioned tables in DBA_TAB_HISTOGRAMS and DBA_TAB_COL_STATISTICS.

SQL*Loader and Partitioned Objects

- Partitioned tables can be loaded using conventional path.
- Partitioned tables can be loaded sequentially using direct path.
- A single table partition can be loaded in parallel using direct path.

SQL*Loader and Partitioned Objects

SQL*Loader can perform the following tasks on partitioned objects:

- Load a single partition or subpartition of a partitioned table. This can be done by specifying the partition or subpartition extended table name in the `INTO TABLE` clause.
- Load all partitions of a partitioned table

SQL*Loader has the flexibility to handle operations on partitioned objects using conventional path, direct path, and parallel direct path.

SQL*Loader Conventional Path

Partitioned tables can be loaded using conventional path:

- **The load uses SQL `INSERT` statements.**
- **The mapping of rows to a partition or subpartition is handled transparently by SQL.**
- **You can run multiple loads on the same table concurrently.**

5-10

Conventional Path

The load uses `SQL INSERT` statements, which distribute the input rows to the correct partitions and update both local and global indexes.

- You can run multiple loads on the same table concurrently.
- You can load a single table partition via the conventional path.
- You must specify the table name and the partition name in the load control file.
- Rows that do not belong to that partition are written to the `BADFILE` file.
- You can load different partitions in the same table concurrently.

SQL*Loader Direct Path Sequential Loads

You can sequentially load a partitioned table through the direct path:

- **Indexes are built automatically.**
- **When loading a direct path in a single partition:**
  - **Local indexes can be maintained by the load.**
  - **Global indexes cannot be maintained by the load.**

5-11

Direct Path Sequential Loads

If you use the direct path load:

- Indexes are built automatically.
- You must specify the table name and the partition name and set `DIRECT = TRUE`.
- If there are no global indexes, you can run sequential direct path loads on different partitions of the same table concurrently.
- Referential integrity and check constraints must be disabled.
- Triggers must be disabled.

SQL*Loader Direct Path Parallel Loads

You can parallel load a single table partition through the direct path:

- **You must specify the table name and the partition name, and:**
  - set `DIRECT = TRUE`
  - set `PARALLEL = TRUE`
- **The corresponding partition in each local index is marked `UNUSABLE`.**
- **There must be no global indexes on the table.**
- **You can run parallel direct path loads on different partitions of the same table concurrently.**

Direct Path Parallel Loads

You can parallel load a single table partition using direct path.

- You must specify the table name and the partition name and set `DIRECT = TRUE` and `PARALLEL = TRUE`.
- The corresponding partition in each local index is marked `UNUSABLE`. You must rebuild the partitions explicitly after the load completes.
- There must be no global indexes on the table. You must drop them before loading and re-create them after the load completes.
- You can run parallel direct path loads on different partitions of the same table concurrently.

Parallel direct path loads are used for intrasegment parallelism. Intersegment parallelism can be achieved by concurrent single partition direct path loads, with each load session loading a different partition of the same table. When loading a parallel direct path in a single partition, consider that neither local or global indexes can be maintained by the load.

Export and Import

You can export or import one or more specified partitions or subpartitions within a table using the partition or subpartition name.

```
exp hr/hr TABLES=(hr.orders:q1_h1, \
hr.orders:q1_h2,hr.employees,sales:p1)
```

```
imp hr/hr TABLES=(hr.orders:q1_h1, \
hr.orders:q1_h2,hr.employees,sales:p1)
```


Export and Import

In this example, ORDERS is a composite partitioned table, and Q1_H1 and Q1_H2 could be either a partition or a subpartition. If Q1_H1 and Q1_H2 are partitions, all of the subpartitions are exported.

The HR.EMPLOYEES table can be a partitioned or nonpartitioned table. The SALES table, however, must be a partitioned table, and P1 must be one of its partitions or subpartitions.

Import provides the following additional option:

- Import creates a composite partitioned table if the exported table was composite partitioned.
- Subpartition export is supported only in Table mode.
- You must specify the *table name:subpartition name.*

Export

```
Export: Release 9.0.1.0.0 - Production on Thu Jan 3 06:42:33 2002
     Format:  EXP KEYWORD=value or KEYWORD=(value1,value2,...,valueN)
     Example: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)
               or TABLES=(T1:P1,T1:P2), if T1 is partitioned table
Keyword    Description (Default)      Keyword      Description (Default)
--------------------------------------------------------------------------
USERID     username/password          FULL         export entire file (N)
BUFFER     size of data buffer        OWNER        list of owner usernames
FILE       output files (EXPDAT.DMP)  TABLES       list of table names
COMPRESS   import into one extent (Y) RECORDLENGTH length of IO record
GRANTS     export grants (Y)          INCTYPE      incremental export type
INDEXES    export indexes (Y)         RECORD       track incr. export (Y)
DIRECT     direct path (N)            TRIGGERS     export triggers (Y)
LOG        log file of screen output  STATISTICS   analyze objects
(ESTIMATE)
ROWS       export data rows (Y)       PARFILE      parameter filename
CONSISTENT cross-table consistency    CONSTRAINTS  export constraints (Y)
...
TTS_FULL_CHECK       perform full or partial dependency check for TTS
VOLSIZE              number of bytes to write to each tape volume
TABLESPACES          list of tablespaces to export
TRANSPORT_TABLESPACE export transportable tablespace metadata (N)
TEMPLATE             template name which invokes iAS mode export
```


Exporting Partitioned Objects

Export provides the following options:

- Table export is supported in all modes (Full, User, Table).
- Partition export is supported only in Table mode.
- You must specify the *table name:partition name.*
- The keyword `QUERY` allows you to select a subset of rows from a table while performing a table mode export. The value of the query parameter is a string that contains a `WHERE` clause for a SQL select statement that will be applied to all tables (or table partitions) listed in the `TABLE` parameter. For example, if user HR wants to export only those employees in department 10, he could do the following:

```
exp hr/hr tables=employees query="where department=10"
```

- Export supports writing to multiple export files and Import can read from multiple export files. If you specify a value (byte limit) for the `FILESIZE` parameter, Export will write only the number of bytes you specify to each dump file. When the amount of data Export must write exceeds the maximum value you specified for `FILESIZE`, it will get the name of the next export file from the `FILE` parameter.

Import

```
Import: Release 9.0.1.0.0 - Production on Thu Jan 3 06:42:49 2002
     Format:  IMP KEYWORD=value or KEYWORD=(value1,value2,...,valueN)
     Example: IMP SCOTT/TIGER IGNORE=Y TABLES=(EMP,DEPT) FULL=N
               or TABLES=(T1:P1,T1:P2), if T1 is partitioned table
Keyword  Description (Default)       Keyword      Description (Default)
--------------------------------------------------------------------------
USERID   username/password           FULL         import entire file (N)
BUFFER   size of data buffer         FROMUSER     list of owner usernames
FILE     input files (EXPDAT.DMP)    TOUSER       list of usernames
SHOW     just list file contents (N) TABLES       list of table names
IGNORE   ignore create errors (N)    RECORDLENGTH length of IO record
GRANTS   import grants (Y)           INCTYPE      incremental import type
INDEXES  import indexes (Y)          COMMIT       commit array insert (N)
ROWS     import data rows (Y)        PARFILE      parameter filename
...
TOID_NOVALIDATE   skip validation of specified type ids
COMPILE           compile procedures, packages, and functions (Y)
VOLSIZE           number of bytes in file on each volume of a file on tape
The following keywords only apply to transportable tablespaces
TRANSPORT_TABLESPACE import transportable tablespace metadata (N)
TABLESPACES tablespaces to be transported into database
DATAFILES datafiles to be transported into database
TTS_OWNERS users that own data in the transportable tablespace set
```


Import and Partitioned Objects

Import provides the following options:

- You can import all the data of a partitioned or nonpartitioned table from a dump file into a partitioned or nonpartitioned table.
- Import creates a partitioned table if the exported table was partitioned.
- If a table is partitioned, Import rejects any rows that fall above the values specified by `VALUES LESS THAN` in the highest partition.
- Import is supported in all modes (Full, User, Tables).
- Partition import is supported only in Table mode.
- You must specify the *table name:partition name*.
- You can skip maintenance of unusable indexes using `SKIP_UNUSABLE_INDEXES`.

You can use `TOID_NOVALIDATE` to specify object types to exclude from TOID comparison.When you import a table that references a type, but a type of that name already exists in the database, Import attempts to verify that the preexisting type is in fact the type used by the table. To do this, Import compares the type's unique identifier (TOID) with the identifier stored in the export file, and will not import the table rows if the TOIDs do not match.

Partitioning and Transporting Tablespaces

When transporting a tablespace set:

- Indexes inside the set of tablespaces must be associated with a table contained in the tablespace set.
- A partitioned table must be fully contained in the set of tablespaces.
- The tablespace set you want to copy must contain either all partitions of a partitioned table, or none of the partitions of a partitioned table.
- If you want to transport a subset of a partition table, you must exchange the partitions into tables before copying the tablespace set.

5-16

Partition Considerations for Transportable Tablespaces

To determine whether a set of tablespaces is self-contained, you can invoke the `TRANSPORT_SET_CHECK` procedure in the Oracle supplied package `DBMS_TTS`. You must have been granted the `EXECUTE_CATALOG_ROLE` role (initially signed to SYS) to execute this procedure.

When you invoke the `DBMS_TTS` package, you specify the list of tablespaces in the transportable set to be checked for self-containment. You can optionally specify if constraints must be included. For strict or full containment, you must additionally set the `TTS_FULL_CHECK` parameter to `TRUE`.

The strict or full containment check is for cases that require capturing not only references going outside the transportable set, but also those coming into the set. Tablespace Point-in-Time Recovery (TSPITR) is one such case in which dependent objects must be fully contained or fully outside the transportable set.

Self-Contained Check

Verify that SALES_1 and SALES_2 tablespaces are self-contained:

```
SQL> EXECUTE dbms_tts.transport_set_check \
('sales_1,sales_2', TRUE);
```


Checking For Self Containment

In the example below, it is determined whether tablespaces sales_1 and sales_2 are self-contained, with referential integrity constraints taken into consideration (indicated by TRUE).

```
SQL>EXECUTE dbms_tts.transport_set_check('sales_1,sales_2', TRUE);
```

After invoking this PL/SQL package, you can see all violations by selecting from the `TRANSPORT_SET_VIOLATIONS` view. The following query shows a case in which there are two violations: a foreign key constraint, `dept_fk`, across the tablespace set boundary, and a partitioned table, `jim.sales`, that is partially contained in the tablespace set.

```
SELECT * FROM TRANSPORT_SET_VIOLATIONS;
VIOLATIONS
-------------------------------------------------------------------
Constraint DEPT_FK between table JIM.EMP in tablespace SALES_1 and
table JIM.DEPT in tablespace OTHER
Partitioned table JIM.SALES is partially contained in the
transportable set
```

Online Table Redefinition

With online table redefinition, you can:

- **Modify the storage parameters of the table**
- **Move the table to a different tablespace in the same schema**
- **Add support for parallel queries**
- **Add or drop partitioning support**
- **Re-create the table to reduce fragmentation**

Online Table Redefinition

The mechanism for performing online redefinition is the `DBMS_REDEFINITION` PL/SQL package. Execute privileges on this package are granted to `EXECUTE_CATALOG_ROLE`. In addition to having execute privileges on this package, you must be granted the following privileges:

- `CREATE ANY TABLE`
- `ALTER ANY TABLE`
- `DROP ANY TABLE`
- `LOCK ANY TABLE`
- `SELECT ANY TABLE`

Parallel Execution and Partitioning

- Long-running operations can be divided into smaller operations, and executed in parallel on individual partitions.
- The granule of parallelism is the partition, except for composite partitions, where it is the subpartition.
- There is now limited support for parallelism within a partition.

ORACLE

5-19

Partitioning and Parallelization

Parallel execution uses multiple slave processes working together to execute a single SQL statement. By dividing the work necessary to execute a statement among multiple slave processes, the RDBMS can execute statements more quickly than a single process. Parallel execution can dramatically improve performance for data intensive operations associated with DSS applications and VLDB environments.

Operations on partitioned tables and indexes are performed in parallel by assigning different parallel execution servers to different partitions of the table or index.

Note: For more more information, please refer to the *Oracle9i Data Warehouse Administration* course.

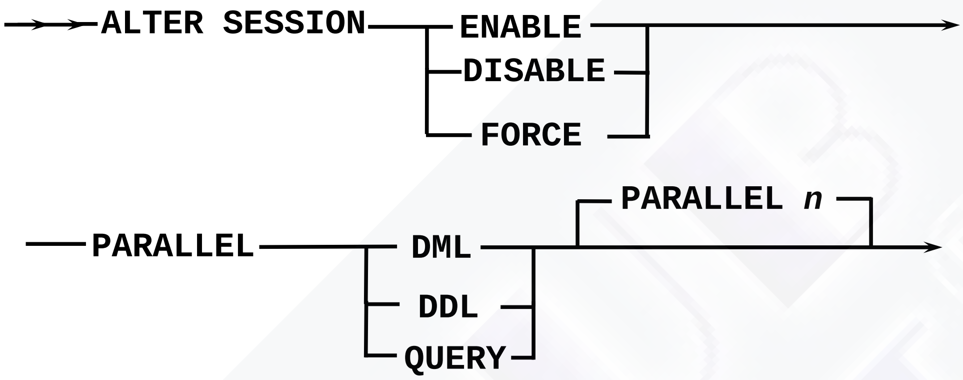
Parallelizable Operations

- DDL statements:
  - `CREATE TABLE AS SELECT`
  - `CREATE INDEX`
  - Rebuild an index
  - Rebuild an index partition
  - Move a partition, split a partition
- DML statements: `UPDATE`, `DELETE`, `INSERT...SELECT`
- Queries: Table scans, nested loops, group by, order by, hash joins, range scan on partitioned index

ORACLE

Enabling Parallel Execution

The `ALTER SESSION` statement enables parallel execution:



5-21

Enabling Parallel DML, DDL, and QUERY

This clause indicates that all subsequent queries, DML, or DDL issued against the RDBMS be considered for parallel execution. It allows the default degree of parallelism of the table to be overridden without changing the tables themselves. This clause can be executed only between committed transactions. Uncommitted transactions must be committed or rolled back prior to executing this clause for DML. You cannot specify the optional `PARALLEL` integer with `ENABLE` or `DISABLE`.

- `ENABLE` executes subsequent statements in the session in parallel. This is the default for DDL and query statements.
- `DISABLE` specifies that subsequent statements are executed serially. This is the default for DML statements.
- `FORCE` forces parallel execution of subsequent statements in the session. If no parallel clause or hint is specified, then a default degree of parallelism is used. This clause overrides any parallel_clause specified in subsequent statements in the session, but is overridden by a parallel hint.

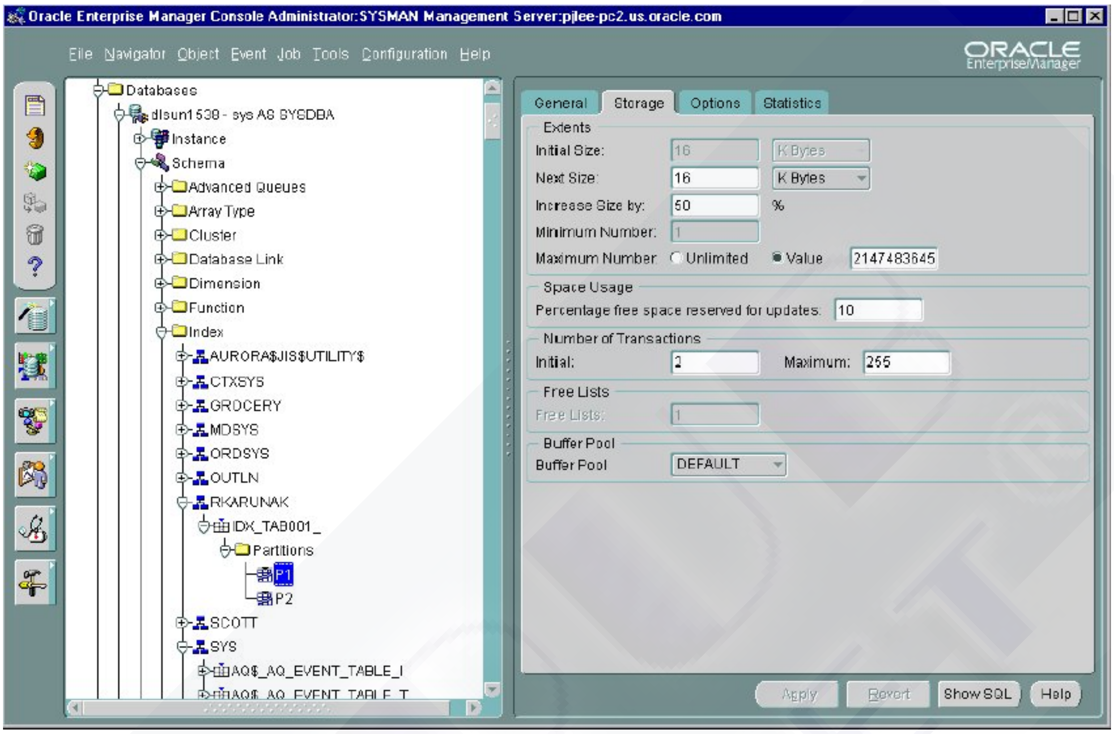
Enabling Parallel DML, DDL, and QUERY (continued)

- `PARALLEL` *`integer`*: Explicitly specifies a degree of parallelism
  - For `FORCE DDL`, the degree overrides any parallel clause in subsequent DDL statements.
  - For `FORCE DML` and `QUERY`, the degree overrides the degree currently stored for the table in the data dictionary.
  - A degree specified in a statement through a hint overrides the degree being forced.

The following types of DML operations are not parallelized regardless of this clause:

- Operations on clustered tables
- Operations with embedded functions that either write or read database or package states
- Operations on tables with triggers that could fire
- Operations on tables or schema objects that object types or LONG or LOB data types
- `UPDATE` or `DELETE` on nonpartitioned tables

OEM Schema Management Window



Schema Management Using OEM

With the Schema Management functionality, you can create, alter, or drop database schema objects such as clusters, indexes, materialized views, tables, and partitioned tables, as well as view dependencies of schema objects.

Autogenerate Range Partitioning Support

Instead of defining a large set of range partitions manually, the Create Table property pages support creating this type of partition automatically. Range partitions can easily be defined by specifying the earliest date or smallest number of those partitions, the length of time or number of each partition, and the total number of partitions. OEM will automatically define and name all partitions, allowing you to create a large number in one simple step.

Summary

In this lesson, you should have learned how to:

- Describe the behavior of partitioned tables and indexes with other database features and utilities
- Describe Oracle Enterprise Manager support of partitioned objects

ORACLE

Practice Overview: Working with Partitioned Tables and Indexes

This practice covers the following topics:

- Exporting and importing a partition
- SQL*loading into range partition table
- Self-containment checking of partitioned objects for transportable tablespaces

6

Practical Partitioning

Objectives

After completing this lesson, you should be able to do the following:

- Achieve a practical understanding of partitioning in practice
- Provide an overview of performance-related issues
- Apply partitioning concepts to real-world scenarios

Areas of Benefit

Partitioning offers benefits in the following areas:

- **Very Large Databases (VLDB)**
  - **Data Warehouses**
  - **Decision Support Systems**
- **Real Application Clusters environments**
- **Parallel execution**

6-3

Areas That Can Benefit From Partitioning

You have seen how the use of partitioned tables and indexes greatly enhances the performance and manageability of very large databases. With partitioned tables, your data can be divided into partitions or even subpartitions. Indexes can be partitioned in similar fashion. Each partition can be managed individually, and can function independently of the other partitions, thus providing a structure that can be better tuned for availability and performance.

Because of the nature of Real Application Clusters architecture, partitioning can be a great aid in the reduction of block contention between instances.

Partitions also provide another method of implementing parallel execution. Operations on partitioned tables and indexes are performed in parallel by assigning different parallel execution servers to different partitions of the table or index.

Applications and Partitioning Strategies

- Database size increases
- Maintain same segmentation strategy
- New application functionality added
  - Accessing disjointed data
  - Should not change current strategy
  - May impact systems using TP monitors
  - Accessing same data
  - May need to revisit partitioning strategy

6-4

Segmentation Strategy

When you design a segmentation strategy, you must consider its applicability to future growth, or you will not have a scalable system. Some types of growth might *not* have a major impact, while others may.

If the database increases in size simply because the tables grow, but for no other reason such as the introduction of new tables or additional user requirements, then the current partitioning strategy should work. In some cases, if you have physically partitioned your tables, you may need to subpartition existing partitions. This is relatively simple and transparent but may require some programming changes if you have built the partitions and related views manually.

If the database is required to support new users and new functions, then the current strategy may not work. In some cases, the new functionality only involves data that is already disjointed based on the current segmentation strategy. In other cases, the strategy may have to be reconsidered and even rebuilt. When it is based on the transaction model, the work to rebuild the system can be very expensive. Even if the same segmentation strategy is still valid, the addition of a new parallel instance can require substantial rewrites to code handling transaction partitioning in order for it to recognize when to use the new instance.

Segmentation: Example

An airline business management system with the following characteristics will help demonstrate segmentation approaches:

- **Phone reservations**
- **Air fleet management**
- **Sales and marketing**
- **Counter operations**

ORACLE

6-5

Example of Segmentation

We will explore some segmentation approaches to see how an international airline could partition the work and the objects related to a system. The basic functions of the database used by this airline consist of the following:

- Phone reservations
- Air fleet management
- Sales and marketing
- Counter operations

Application Partitioning: Step 1

Define the major functional areas of the system:

- **List the basic functions**
- **Group smaller functions into larger functions to avoid too fine-grain components**

6-6

Step 1: Define the Major Functional Areas of the System

Subdivide the major functions of the airline by geography for continuing analysis:

- USA phone reservations
- European phone reservations
- Asia/Pacific phone reservations
- Global air fleet management
- USA sales and marketing
- Non-USA sales and marketing
- USA counter operations
- Europe counter operations
- Africa counter operations
- Asia counter operations
- Australasia counter operations

Application Partitioning: Step 2

Phone Reservations	Counter Operations
Table 1	Table 5
Table 2	Table 6
Table 3	Table 7
Table 4	Table 8
Table 5	Table 9
Table 6	Table 10

Identify table access paths of each application

ORACLE

6-7

Step 2: Identify Table Access Paths of Each Application

In this step, list the tables involved in each application. This task is simplified for the purpose of illustration by selecting just two of the functions of the fictional airline company's system. The diagram lists the tables in order, under each of the functions. Your diagram of the system would probably use alphabetic ordering of the table names.

Application Partitioning: Step 3

Phone Reservations	Overlaps	Counter Operations
Table 1	Table 5	
Table 2	Table 6	
Table 3		Table 7
Table 4		Table 8
		Table 9
		Table 10

Define table overlaps between applications

ORACLE

6-8

Step 3: Define Table Overlaps Between Applications

In this step, identify and list those tables that are used by more than one application. The remaining tables do not need to be considered for further segmentation. If you are planning a distributed database, each application's nonshared tables can be stored in the database for that application. If you are planning to use Real Application Clusters, these tables would be associated with the users of just one instance.

Application Partitioning: Step 4

Phone Reservations	Overlap Access	Overlaps	Overlap Access	Counter Operations
Table 1	Type		Type	
Table 2				
Table 3				Table 7
Table 4	S	Table 5	S	Table 8
	I, U	Table 6	I, U	Table 9
				Table 10

Define access types of overlaps

S Select I Insert U Update

ORACLE

Step 4: Define Access Types of Overlaps

In this step, list the uses of each of the tables that are in the overlap column; that is, tables that are shared by two or more applications in the system. You can use the terminology shown in the diagram (S for Select, I for Insert, U for Update, and D for Delete).

This step highlights the tables that are shared by applications only for query purposes. These tables are easier to maintain in a distributed database even if they are needed in each local database. The lack of changes means that once they have been replicated to each database, they do not require ongoing maintenance to keep them synchronized. Similarly, in Real Application Clusters environments, the read-only tables can be shared easily by the users on each instance as long as they are separated by tablespace from active tables.

Application Partitioning: Step 5

Phone Reservations	Overlap Access	Overlaps	Overlap Access	Counter Operations
Table 1	Type &		Type &	
Table 2	Volume		Volume	
Table 3				Table 7
Table 4	S (10/s)	Table 5	S (50/s)	Table 8
	I (100/s)	Table 6	I (10/s)	Table 9
	U (50/s)		U (90/s)	Table 10

Identify transaction volumes of overlaps

Step 5: Identify Transaction Volumes of Overlaps

During this step, use statistics from your current database or estimates of activity for your scalable database. The purpose is to assign a transaction rate for each of the access types defined for the overlapping tables in Step 4. Such rate information will help identify which tables might cause problems with keeping the information synchronized between multiple databases or multiple instances.

Application Partitioning: Step 6

- Classify the overlaps
- Ignore nonoverlapping tables
- Ignore select-only overlaps
- Ignore low-frequency overlaps
- Categorize insert-only tables and their indexes
- Categorize mixed access tables and their indexes

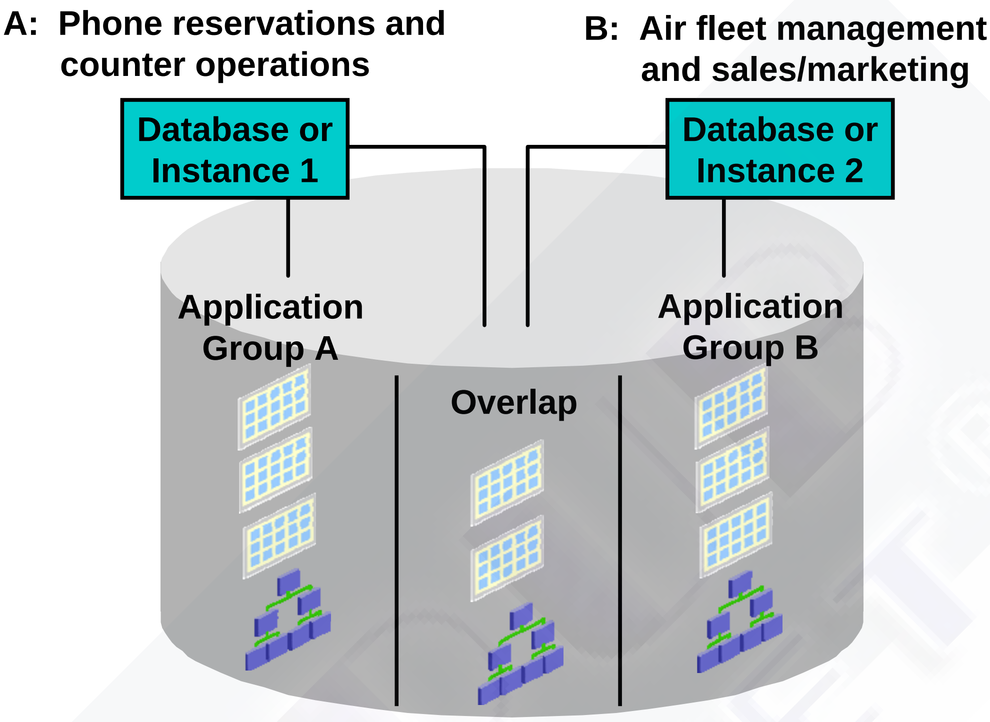
Step 6: Classify the Overlaps

In this step, formalize the results of the previous steps. Segregate the various types of tables based on whether or not they are shared across applications, and then categorize the shared tables. Identify those tables that are used purely for queries. Define the remaining tables as low activity or high activity. You can categorize a table as low activity if it is involved in no more than a few transactions per second for each application, or used primarily for one application with only a few transactions per minute by the others. The remaining high-activity tables are then separated by whether the access is primarily to store new rows (insert-only tables) or whether it involves all DML (inserts, updates, and deletes).

Depending on how you plan to segment your system, you use these results in different ways. As discussed on the previous pages, nonshared tables and read-only tables are generally not an issue for either segmentation into multiple tablespaces or segmentation into multiple instances using Real Application Clusters. Low activity tables are also easy to handle because these two types of segmentation can manage to keep up with the changes without incurring performance penalties. However, if you have too many tables, even though no single one causes overall system degradation, in combination, they can. It is therefore important to know the size of the load the total number of tables would add to the network for distributed databases or to locking for Real Application Clusters.

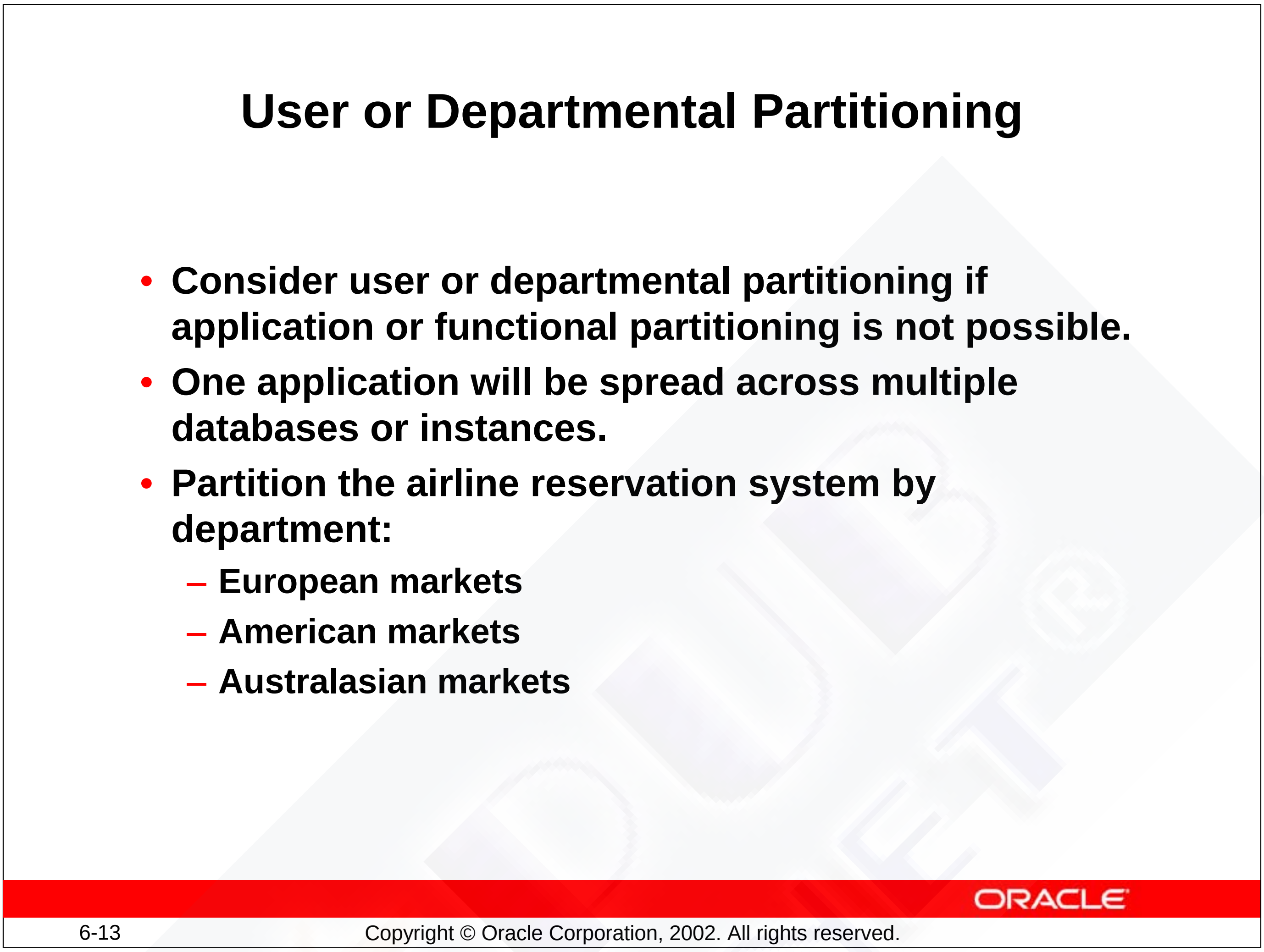
The difference between insert-only and mixed access tables is primarily considered for Real Application Clusters. There are some extent assignment and locking options that can make insert-only tables easier to manage across multiple instances than mixed access tables.

Configuration 1



Segment the Database

In Step 6 of the fictional airline application partitioning, the number of transactions on overlap tables indicated that the two applications, phone reservations and counter operations, were too high to allow them to be segmented. However, similar analyses of the other applications showed that the air fleet management and sales and marketing applications were reasonably independent of the other two applications. This allows us to segment the database either by building two smaller databases or using two Real Application Clusters instances.



About User/Departmental Partitioning

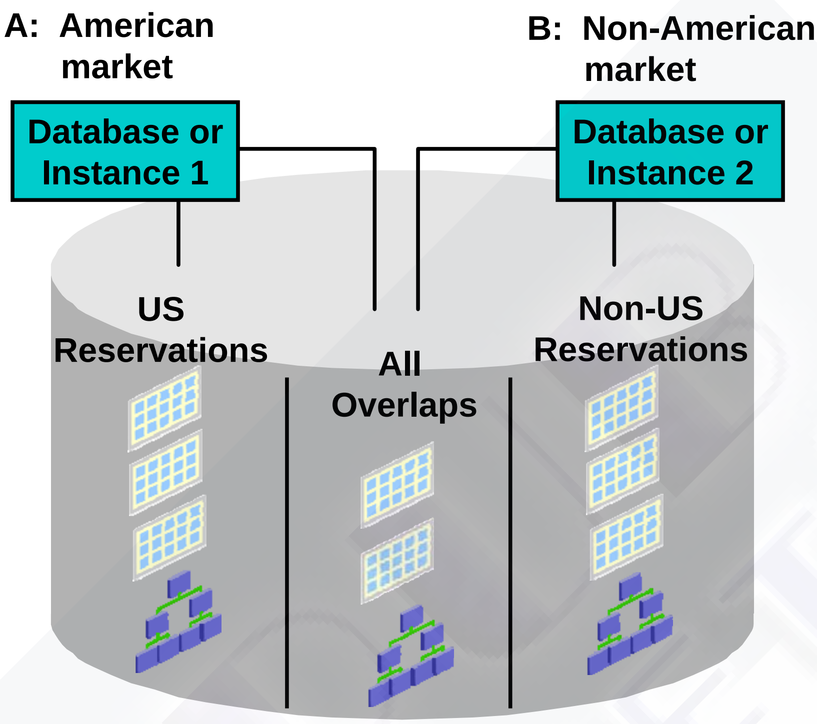
Sometimes it is not possible to segment your system by application. In this case, consider some of the other segmenting options. For example, had the airline been unable to use the configuration shown on the previous slide, it could consider geographic partitioning by national departments. Here is a breakdown by department, or geography in this case, as part of the original application breakdown earlier in this lesson.

In the example, the reservation system needs to be segmented. To achieve this, because it is already one of the major applications, consider user or departmental partitioning.

This approach can be particularly useful when the departments are geographically dispersed because it is a natural segmentation for distributed databases. Each location uses a local database for the data that is partitioned by department. The remaining applications, if any, can be managed from a central database, share one of the departmental databases, or be replicated so that each local database also contains all of the nonlocal data. Combinations of these options are also possible.

Oracle Real Application Clusters technology can also benefit from user or departmental partitioning as long as users are restricted to the instance that is designated to support their department.

Configuration 2



6-14

Segmentation by Department

The segmentation shown in Configuration 2 is based on a departmental breakdown of the reservation application. The segments belonging to the other applications become part of the overlaps and are shared by both of the other segments. If a distributed database approach is used, these overlaps would be in a separate database, in either of the two local databases, or they would be replicated in both local databases.

For a Real Application Clusters environment, the tables are stored in the same database. The overlap segments can cause some interinstance contention and can require a careful strategy when assigning locks.

Application of Partition Types

Partitioning methods offered by Oracle:

- **Range partitioning**
- **Hash partitioning**
- **List partitioning**
- **Composite partitioning**

Range Partitioning

Range partitioning specifics:

- Useful when rows must be mapped to partitions based on ranges of column values
- The data has logical ranges into which it can be distributed, such as months or quarters in a year
- Most efficient when the resulting partitions are of a similar size

ORACLE

6-16

Applying Range Partitioning

Range partitioning is most beneficial when the data has logical ranges into which it can be distributed. A good example is the SALES table partitioned by sales quarters. Performance is optimum when the data evenly distributes into the partitions across the specified range. If the partition size varies greatly, one of the other methods might work better.

Below is a review example of range partitioning:

```
CREATE TABLE sales
   ( invoice_number NUMBER,
     sale_year INT NOT NULL,
     sale_month INT NOT NULL,
     sale_day INT NOT NULL )
  PARTITION BY RANGE (sale_year, sale_month, sale_day)
   ( PARTITION sales_q1 VALUES LESS THAN (2001, 04, 01)
     TABLESPACE ts01,
     PARTITION sales_q2 VALUES LESS THAN (2001, 07, 01)
     TABLESPACE ts02,
     PARTITION sales_q3 VALUES LESS THAN (2001, 10, 01)
     TABLESPACE ts03,
     PARTITION sales_q4 VALUES LESS THAN  (2002, 01,01)
     TABLESPACE ts04 );
```

Hash Partitioning

Hash partitioning is useful:

- **When your data does not easily fit the range partitioning criteria**
- **Because it allows *unbalanced* data to be evenly distributed among the table's partitions**
- **When partition pruning and partition-wise joins on a partitioning key are important**
- **Because the data placement is easily tunable. Partitions can be placed on different partitions, effectively striping your data.**

ORACLE

6-17

The Hash Partitioning Method

Hash partitioning is useful when the data does not easily fit the criteria for range partitioning, but you want to enjoy the performance and manageability benefits provided by partitioning. Rows are mapped into partitions based on a hash value of the partitioning key. The hash function works best with a large number of values. Creating and using hash partitions gives you a highly tunable method of data placement, because you can influence availability and performance by spreading these evenly sized partitions across I/O devices (striping).

Below is an example of the DDL used to create a hash partitioned table:

```
CREATE TABLE washer_lots
 (id NUMBER,
  name VARCHAR2 (60))
PARTITION BY HASH (id)
PARTITIONS 4
STORE IN (lot1, lot2, lot3, lot4);
```

In the example above, each of the four partitions will be stored in a different segment.

List Partitioning

- List partitioning allows precise control of how the rows will map to the partitions.
- Provides a method for unordered or unrelated sets of data to be easily grouped and organized together.
- List partitioning does not support multicolumn partition keys.

List Partitioning Applied

List partitioning allows explicit control when mapping rows to partitions. You can specify a list of unrelated or discrete values for the partitioning column in the description for each partition. Data that is random or unordered can be easily partitioned using this method. The example below illustrates this principle. The table `SALES_BY REGION` is partitioned by six different lists of states. The only relationship to each other is their proximity to one another on the map.

```
CREATE TABLE 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', 'ID'),
 PARTITION q1_southwest VALUES ('AZ', 'UT', 'NM'),
 PARTITION q1_northeast VALUES ('NY', 'VM', 'NJ', 'ME', 'DE'),
 PARTITION q1_southeast VALUES ('FL', 'GA', 'AL', 'MS'),
 PARTITION q1_northcentral VALUES ('ND', 'SD', 'WI'),
 PARTITION q1_southcentral VALUES ('OK', 'TX', 'LA'));
```

Composite Partitioning

- Range partitioning is used to partition the data, and the partitions are subpartitioned using the hash method.
- Well-suited for historical data
- Data manageability is enhanced because this method is ideal for striping applications.
- Data is easy to isolate so that operations on the base table can be parallelized readily.

ORACLE

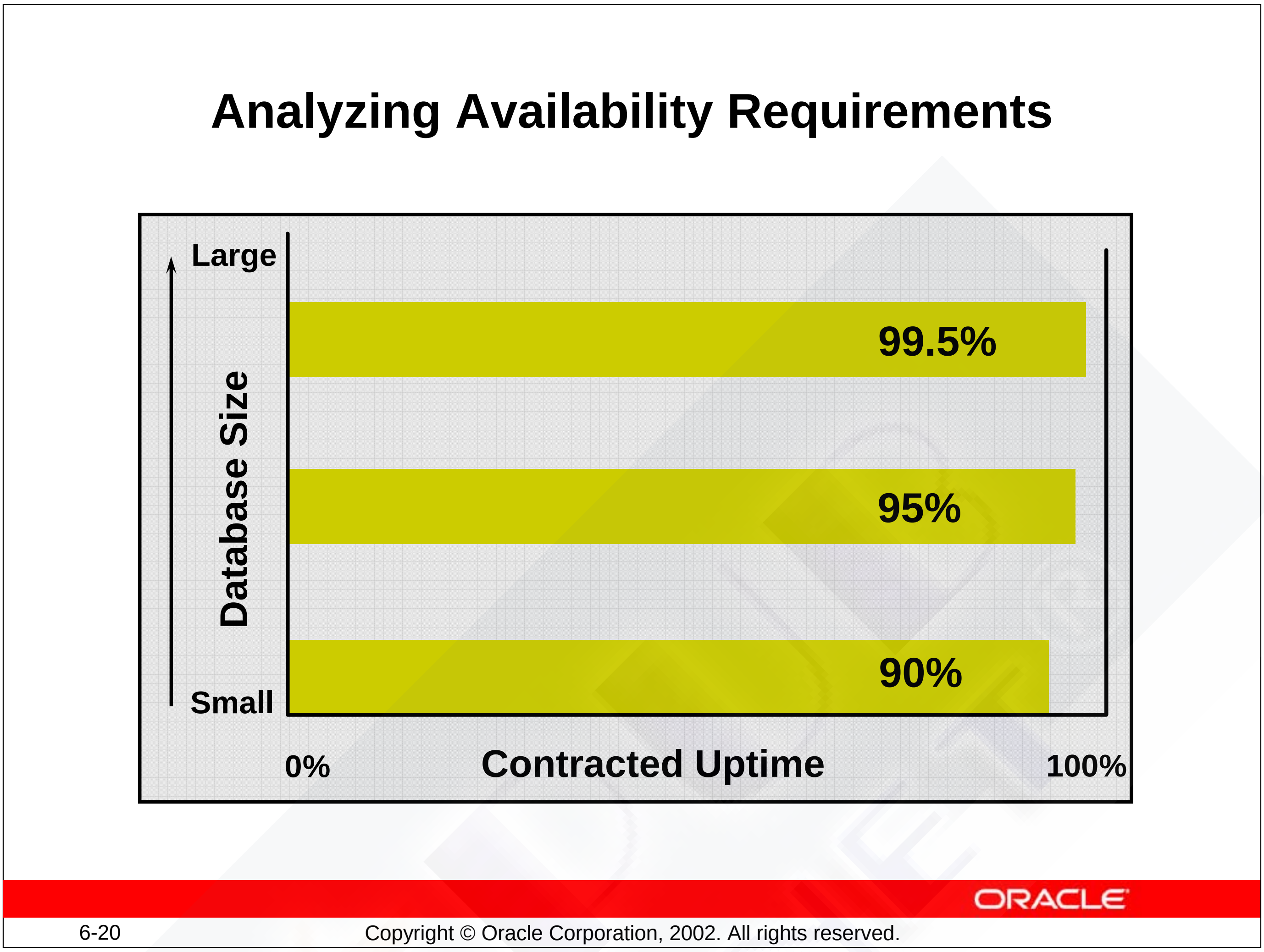
6-19

Applying Composite Partitioning

This method initially partitions the data by range. Each partition is further subpartioned using the hash method. This approach allows for a finer granularity of control over where the data will be located physically. Striping large amounts of data over many devices is more straightforward when composite partitioning is used. Because of the placement of the data, parallelizing operations can greatly enhance performance. This method is well suited for data warehouses and large decision support systems. The example below illustrates this two-part partitioning approach:

```
CREATE TABLE scubagear
(equipno NUMBER, equipname VARCHAR(32),price NUMBER)
 PARTITION BY RANGE (equipno) SUBPARTITION BY HASH(equipname)
 SUBPARTITIONS 8 STORE IN (ts1, ts2, ts3, ts4)
 (PARTITION p1 VALUES LESS THAN (1000),
  PARTITION p2 VALUES LESS THAN (2000),
  PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

In this example, the subpartitions are not explicitly named, so the system will assign them upon table creation.



Dealing With Very Large Databases

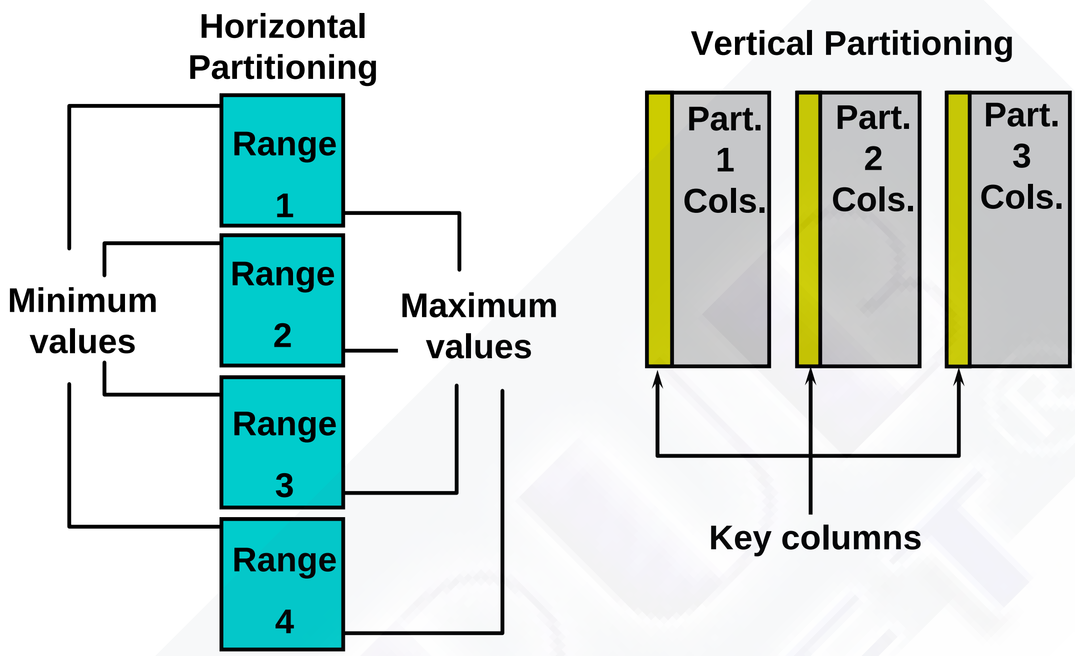
As databases grow in size, the amount of time required to perform certain maintenance and management activities increases too. In some cases, these activities can exceed the time allocated for them. Scheduling flexibility suffers.

In many cases, the DBA is required to meet an agreed-to service level agreement to keep the database available for a given percentage of time. Unfortunately, as the database scales up to become a very large database (VLDB), more and more components are introduced which increases the possibility of a failure.

Some of the specific activities that tend to require more time as the database size increases include:

- Batch job processing
- Archiving and removing old data
- Software enhancements
- Preparing data to be moved to a data warehouse from a data mart
- Download from a data warehouse to data marts
- Data loads or unloads
- Hardware upgrades

Horizontal and Vertical Table Partitions



Horizontal Partitioning

When a table is horizontally partitioned, each complete row is stored in a specific partition based on some predefined characteristic. Most commonly, the value of a column or set of columns is used to define the boundaries of each partition. The row is stored in a specific partition based on the value of its key columns or a value derived by hashing this value according to an internal algorithm.

If you use horizontal partitioning, you can query the whole table with a group function, such as a UNION, executed implicitly or explicitly, depending on how the partitioning was defined. If you are managing the partitions manually, you must make sure that rows are only allowed in their designated partitions, and that updates to the key columns do not result in rows being placed in the wrong partition.

Vertical Partitioning

A vertically partitioned table is divided so that some columns are in one partition and other columns are in another partition. In order to see the table as a whole, a common key value that is unique for each logical row must be stored in each of the partitions.

Vertical Partitioning (continued)

This allows a standard join operation to connect the row pieces from each partition when needed. A vertically partitioned table requires more space to store than a horizontally partitioned table because the key must be stored in every partition repeatedly.

Vertical partitioning is useful when a subset of columns (those related to salary and compensation in an employee table, for example) is queried and processed by one set of users while the remaining columns are more generally available.

Collecting Statistics for Partitioned Objects

- Statistics can be gathered at various levels:
  - Table or Index
  - Partition
  - Subpartition
- Statistics are considered to be global or nonglobal.
- The DBMS_STATS package can gather global statistics at any level for tables only.
- Global histograms and global statistics for indexes cannot be gathered.

ORACLE

6-23

Cost-Based Optimization and Partitioned Objects Statistics

Statistics can be gathered by partition or subpartition by using the DBMS_STATS package. The cost-based optimizer is always used for SQL statement accessing partitioned tables or indexes.

For partitioned objects, the Oracle server maintains separate sets of statistics: at the object level, the partition level, and the subpartition level. If a SQL statement accesses only one fragment, then the Oracle server uses the fragment level's statistics. If a SQL statement accesses multiple fragments, the Oracle server uses a single access path for all of these fragments and uses the statistics from the next higher level. (Here, a fragment can be a subpartition or a partition of a noncomposite object; subpartition is considered as the lowest level, partition is the next level, and object is the last.)

DBMS_STATS always collects global statistics (except histograms and indexes) at any level and never merges them. Global statistics are obtained by considering multiple fragments as only one.

DBMS_STATS Examples

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS( -
ownname => 'SH', tabname => 'sales', -
partname => 'SALES_Q1_2000', -
granularity => 'partition')
```

```
EXECUTE DBMS_STATS.GATHER_INDEX_STATS( -
ownname => 'SH', indname => 'sales_time_bix' )
```


The `DBMS_STATS.GATHER_TABLE_STATS` Procedure

The following list contains some of the important arguments used by this procedure:

- `ownname`: Schema of table to analyze
- `tabname`: Name of table
- `partname`: Name of partition or subpartition depending on granularity
- `method_opt`: Equivalent to the `FOR` clause of the `ANALYZE` command for histograms
- `degree`: Degree of parallelism (`NULL` means use table default value.)
- `granularity`: Granularity of statistics to collect (only pertinent if the table is partitioned)
  - `DEFAULT`: Gather table- and partition-level statistics
  - `SUBPARTITION`: Gather subpartition-level statistics
  - `PARTITION`: Gather partition-level statistics
  - `GLOBAL`: Gather object-level statistics
  - `ALL`: Gather all (subpartition-, partition-, and object-level) statistics
- `cascade`: Gather statistics on the indexes for this table. Index statistics gathering is not made parallel.

Note: The arguments listed above should be considered a partial list. Refer to the *Oracle9i Supplied PL/SQL Packages Reference* for a complete treatment.

The DBMS_STATS.GATHER_INDEX_STATS Procedure (continued)

This procedure gathers index statistics. It does not execute in parallel. Below are some important arguments for this procedure:

- `ownname`: Schema of index to analyze
- `indname`: Name of index
- `partname`: Name of partition or subpartition
- `estimate_percent`: Percentage of rows to estimate (`NULL` means compute.)
- `stattab`: User stat table identifier describing where to save the original statistics
- `statid`: Identifier (optional) to associate with these statistics within `stattab`
- `statown`: Schema containing `stattab` (if different than `ownname`)

Parallel Index Scans

- Partitioned indexes are scanned in parallel by assigning each slave a different partition of the index to scan.
- The number of parallel query slaves is limited by the number of partitions.
- No more than one slave per partition is assigned.

6-26

Parallelization of Index Scans

Operations against tables with partitioned indexes that cause a full index scan can benefit from parallelization. The scan operation can be be divided among several parallel slave processes. No more than one process per index partition can be assigned. Consider the following select:

```
select /*+  parallel(sales_idx2,3) */ * from sales \
where name  = 'MHARTSTEIN';
```

In the example above, there are three slaves working on three partitions. Each query slave is working on individual partitions. The assignment would look like this:

Query_slave_1 => index_partition_1
Query_slave_2 => index_partition_2
Query_slave_3 => index_partition_3

Summary

In this lesson, you should have learned how to:

- Achieve a practical understanding of partitioning in practice
- Provide an overview of performance-related issues
- Apply partitioning concepts to real-world scenarios

Practice Overview: Partitioning Applications

This practice covers the following topics:

- Perform a rolling window operation
- Converting partitioned views to partitioned tables

A
Practices

Lesson 1 Practices

1. Prior to the introduction of the Oracle Partitioning option, manual partitioning was performed to address large table manageability. Can you list some difficulties that might be encountered when using manual partitioning?

   ____

   ____

   ____

2. The Oracle Partitioning option offers many advantages to the database administrator when dealing with very large tables and indexes. Can you list some of these advantages?

   ____

   ____

   ____

   ____

   ____

   ____

3. Please explain the concept and benefits of partition pruning.

   ____

   ____

   ____

4. Can you think of a situation where it would be beneficial to partition the index rather than the associated table?

   ____

   ____

   ____

5. List the four partitioning methods and briefly explain each one.

Practice 2 General Comments

Log in to the `DATAMGR` schema, using the password `DATAMGR` for these practices unless otherwise noted.

Because the creation commands are rather lengthy, it is recommended that you use scripts to make it easier to re-create the tables with variations.

Practice 2-1 Solution: Create Partitioned Tables of Each Type

1. Create a range-partitioned table. The table contains sales history data, and is partitioned by quarters. The table uses rolling window operations, where a new quarter is added, an old one is dropped.

   Table Structure: Name: SHIPPED. Columns: PROD_ID, CUST_ID, DATETIME, UNITS_SOLD, AMOUNT_SOLD, the data type of all columns is NUMBER, except the DATETIME which is TIMESTAMP WITH LOCAL TIME ZONE.

   Partition: Range partion on DATETIME. The partitions for each quarter are to be named SHP_Qn_yyyy where "n" is 1 to 4 and "yyyy" is the year. The first partition is SHP_Q1_2002; the last one is SHP_Q2_2003, 6 partitions. Each partition should contain the appropiate rows based on the DATETIME value.

   Storage: The partitions for year 2002 go to tablespace DATA02, the year 2003 into DATA03, and so on. Initially, you are loading bulk data so all partitions must have PCTFREE 5, except the last one which needs PCTFREE 20. Other storage attributes can be default values.

   Note: To specify a TIMESTAMP literal, the syntax is
   `TIMESTAMP 'yyyy-mm-dd hh:mm:ss.ff +hh:mm'`
   This syntax is fixed, invariant of the NLS settings.

2. Examine the data dictionary views to verify that the partition definitions and storage attributes are defined correctly.

3. Create a list-partitioned table. The table contains customer addresses that include a country code, and should be partitioned by continent. This is in anticipation of the need for extensive data manipulation, which will occur region-by-region.

   Table Structure: Name: CUSTS. Columns and datatype: CUST_ID NUMBER, FIRSTNAME VARCHAR2(20), LASTNAME VARCHAR2(40), ADDRESS VARCHAR2(20), COUNTRY CHAR(2).

   Partition: List partition on `COUNTRY`.

Partition name (Region)	Partition Key (Country codes)	Comment
CUST_AM	US, CA	North America
CUST_SA	AR, BR	South America
CUST_EU	DE, FR, UK, DK, ES, IE, NL	Europe
CUST_XX	AU, IN, JP, MY, NZ	Others

   NULL is to be an allowed value.

Storage: All of the partitions are to be stored in DATA04. Other storage attributes can be default values.

4. Examine the data dictionary views to verify the partition key values.

5. Create a hash-partitioned table. The hashing is required to ease management operation with the table.

   Table Structure: Name: TEST_RESULT. Columns: TEST_ID NUMBER, BATCH_NO NUMBER, RESULT_A VARCHAR2(4000), RESULT_B VARCHAR2(4000).

   Partition: HASH partion on TEST_ID into 8 partitions. The partition names are irrelevant.

   Storage: The partitions are to be evenly spread in tablespaces DATA01, DATA02, and DATA03. Partition segments must have the initial segment size of 200K.

6. Examine the data dictionary views to verify that the consumed storage amount is correct. **Note**: This will differ from the specified initial extent size, if the tablespace is locally managed with Automatic or Uniform Extent Size, or has had Minimum Extent defined.

7. Create a range-hash composite partitioned table. The SHIPPED table from step 1 above needs subpartitioning. Because you want to use the DBMS_REDEFENITION package to migrate, you will create the new table structure as a first step. (The DBMS_REDEFINITION, which allows for table restructuring while the data, including updates, remains available for users, will not be covered in this course.)

   Table Structure: Table name is SHIPPED_T. Table structure is the same as for SHIPPED. You can choose to use CREATE TABLE … SELECT AS … WHERE ROWNUM<1 to copy the structure.

   Partition: Range partition is the same as for the SHIPPED table.. Subpartition on columns CUST_ID and PROD_ID. Four subpartitions per partition. Names are irrelevant. Allow for large changes in the DATETIME value.

   Storage: Subpartition segments to be stored in DATA02 for the year 2002, and DATA03 for the year 2003, except for the last partition (SHP_Q2_2003), which needs the subpartitions stored in DATA01 and DATA04.

8. Use the Data dictionary view to determine the tablespace in which the subpartitions are stored.

9. The TEST_RESULT table needs to be re-created. The result text is too large for VARCHAR2(4000) and rather than add another RESULT_3 and so on, a CLOB will be used. Fortunately, the data can be reloaded so that you can drop and recreate the table.

   Table structure: As above, but instead of RESULT_1 and RESULT_2, use a column named RESULTS of type CLOB.

   Partition: No change to the table partition specification for TEST_RESULT from earlier.

Storage: The table partitions are all to be stored in DATA04 without specifying any initial sizing. The CLOB partitions are to be spread in DATA01, DATA02 and DATA03.

10. Check the tablespace allocation. Hint: All partition names of this table start with ‘H’.

Note: At the end of this practice, you should have four tables in the DATAMGR schema, that will be used in subsequent practices.

Practice 2-2 Use the data dictionary to verify the partition structure

1. Create a “plain vanilla” table. Name: VANILLA. Columns: DATA of NUMBER and TEXT of VARCHAR2(20). No partitioning. Place in tablespace USERS.

2. Display the table name, partition type, and row movement status of the tables created so far. The nonpartitioned table should be identified as such.

3. Display the table names that have some table partition in tablespace DATA03. Ignore LOB segments.

4. Display the partition columns used for SHIPPED and SHIPPED_T.

5. Log in to the SH sample schema, password SH.

6. Find which tables are partitioned. Determine how many rows are contained in one of the partitioned tables, and in one of the table’s partitions.

7. Log in again to the DATAMGR schema.

Note: At the end of this practice, you should have 5 tables in the DATAMGR schema.

Practice 2-3 See row placement in partitions

1. Insert a few rows into the CUSTS table. For example, 3 rows with a customer residing in the countries US, CA, and DE, respectively.

2. Select the block number of each row's rowid. The function DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid) accomplishes this.

3. Populate the table with data from the SH schema's CUSTOMER table, using

   ```
   INSERT INTO custs
     SELECT cust_id, cust_first_name, cust_last_name,
            cust_street_address,country_id
     FROM sh.customers ;
   ```

   The insert should fail. Select an appropriate subset of the data so the insert succeeds, and commit the insert.

4. As SYSTEM/MANAGER, examine the DBA_EXTENTS view to determine which blocks belong to a particular partition segment of the CUSTS table. Compare this to the DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) returned from a query of some customers, for example, all customers in CUSTS with CUST_ID less than 200, and verify that rows are placed in the right partition.

   Log back in to the DATAMGR schema after completing this exercise.

Practice 2-4 Verify Partitioning Pruning Takes Place

1. Create the table `PLAN_TABLE` by executing the `utlxplan` standard script. **Note**: This is located in ORACLE_HOME/rdbms/admin.

2. Use the `EXPLAIN PLAN` statement to find the execution plan for:
   - a query of the whole of the SHIPPED table
   - a query of named partition of the SHIPPED table
   - a query of a range of values in the DATETIME column of the SHIPPED table, for example 1-JUN-2001 to 31-AUG-2001.
   - A query of some countries from the CUSTS table, using a IN list, for example `COUNTRY IN ('DE', 'FR', 'UK')`

   Use the `SET STATEMENT_ID` clause to distinguish your separate plans, or `TRUNCATE` the PLAN_TABLE between each execution plan.

3. View the execution plans. You need only view the columns STATEMENT_ID, OPERATION, PARTITION_START, and PARTITION_STOP from the table PLAN_TABLE.

4. Populate the SHIPPED table with sample data from the SH schema's SALES table.

```
INSERT INTO shipped
  SELECT prod_id, cust_id, (3*365)+time_id,
         quantity_sold, amount_sold
  FROM sh.sales
  WHERE time_id between '3-JAN-1999' and '30-JUN-2000'
  and   cust_id < 10000
  and   prod_id < 5000  ;
```

   The command is available in
   `$HOME/STUDENT/LABS/lab_02_04_populate_shipped.sql`.

5. Repeat the execution plan from above for the SHIPPED table. Use a suffix on the STATEMENT_ID to distinguish the first round from this round. Check the execution plans now.

6. OPTIONAL

   Repeat step 4 from the practice 2-3 to verify correct placement of rows in the SHIPPED table.

   Remember to connect back into the DATAMGR schema.

Practice 3-1 Create most types of partitioned index

1. Create a normal, nonpartitioned index on the partitioned SHIPPED table.

   Index: Name SHP_NP_CI. Index the column CUST_ID.

   Storage: Default

2. Create a global partitioned index on CUSTS table.

   Index: Name CST_GL_LFN. Index on the columns LASTNAME, FIRSTNAME.

   Partition: Range partition. Try first the FIRSTNAME column as the partition key. Name the partitions: C_G_1, C_G_2, C_G_3 with end values 'A', 'G' and MAXVALUE, respectively.

   Storage: Use tablespaces INDX01, INDX02, and INDX03, one for each.

   Why will the index creation fail?

   ____

   Do it with the LASTNAME column as the partition key, but otherwise use the same definition.

3. Create a local index on the partitioned CUSTS table.

   Index: Name CST_LC_FN. Index on the column FIRSTNAME.

   Partition: Partition names are irrelevant.

   Storage: Use tablespace INDX04.

3b: Could you have specified anything else about the partition type or partition key values?

   ____

4: Without referring to the USER_PART_INDEXES view, but possibly by examining other views, determine if this local index is prefixed or not. Afterwards, check your answer by selecting from USER_PART_INDEXES.ALIGNMENT.

5. Create a local partitioned index on SHIPPED.

   Index: Name: SHP_LC_PI. Normal index on column PROD_ID

   Storage: Specify nothing, use all defaults.

6. Where should the local index partitions be stored: in the user default, the table default, or the current table partitions storage? After considering your answer, check it in the data dictionary.

   ____

7. Create a global index on SHIPPED.

   Index: Name SHP_GL_AM. Index on the column AMOUNT.

   Partition: Range partition. Only one possible column can be the partition range key. Name the partitions: S_G_1 and S_G_2, with the partition key value for the first partition at 10. There is only one workable value for the second partition key value.

   Storage: Use tablespaces INDX01 and INDX02.

8. Create a partitioned local bitmap index on SHIPPED.

   Index: Name: TST_LB_TI. Type: Bitmap. Index the columns TEST_ID.

   Storage: All defaults.

9. A bitmapped global partitioned index is attempted on the hash-partitioned table TEST_RESULT. Will it succeed, and if not what is the failure reason? Try it.

   Index: Name TST_LB_BN. Type: Bitmap: Index on the column BATCH_NO.

   Partition: Range partition. Partitions to be named T_G_1 and T_G_2, with the partition key value for the first partition at 10.

   Storage: Use tablespaces INDX01 and INDX02.

   ____

10. Use the data dictionary views to verify that your indexes are partitioned as expected. List partition key values and tablespace used as appropriate.

Practice 3-2: Specifying partitioned constraints

1. Attempt to add a unique key constraint, CST_LUQ_CI, to the CUSTS table on the CUST_ID column. The unique index created to support the constraint is to be local partitioned. Why does this fail?

   ____

2. Extend the unique constraint definition so it can be locally partitioned.

3. Examine the partition names and storage location.

4. Create a global partitioned constraint on the hash-partitioned TEST_RESULT table.

   <u>Constraint and supporting index:</u> Name: TST_GUQ_BN. Column: BATCH_NO.

   <u>Partitioning:</u> Range partitioned on BATCH_NO, partition key values at 100, 200 and MAXVALUE

   <u>Storage:</u> Defaults

Practice 4-1 Drop and Add table partition, with index maintenance

1. Another season has gone by, and it is time to do the Rolling Window Operation on the SHIPPED table. Drop the SHP_Q1_2002 partition., without any index maintenance.

2. Examine the index status of all indexes on SHIPPED.

3. Attempt the following insert.

```
INSERT INTO shipped VALUES
   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 1234 ) ;
```

4. Fix the global index invalid status preventing the insert, and then attempt the insert again.

5. Instead of fixing all partitions of the index in the last error message, only rebuild the S_G_2 partition.

6. Attempt the above insert again. Attempt it also with the value 2.22 in the last column, AMOUNT. Commit the successful insert.

7. Check if the same partial index errors occur on queries. Query the table twice on AMOUNT having values 1234 and 2.22, respectively.

8. Fix any remaining indexes so the insert with the value 2.22 also succeeds and commit it.

9. Having dropped and discarded the old data in step 1 above, you must make room for the new data. Add another partition to the SHIPPED table, continuing the pattern of partition attributes.

10. Examine index status.

11. Make the following inserts. Note the date or quarter of each insert.

```
INSERT INTO shipped VALUES
 ( 2847, 5190, TIMESTAMP '2002-02-02 00:00:00.00', 1, 1234 ) ;

INSERT INTO shipped VALUES
 ( 2847, 5190, TIMESTAMP '2003-09-09 00:00:00.00', 1, 1234 ) ;

INSERT INTO shipped VALUES
 ( 2847, 5190, TIMESTAMP '2003-11-11 00:00:00.00', 1, 1234 ) ;
```

    Which inserts should fail? Which might have an undesirable effect? Commit inserts.

    ____

    ____

    ____

Practice 4-2: Split and merge a partitioned table

1. Examine the table CUSTS. Because the volume of data is too skewed, you decide that the countries need to be rearranged by partition..

```
SQL> SELECT country, COUNT(country)
  2    FROM custs
  3    GROUP BY country ;

 CO COUNT(COUNTRY)
 -- --------------
 CA              2
 US          14172
 AR            253
 BR            759
 DE           8041
 DK            353
 ES           1986
 FR           3751
 IE           1958
 NL           7563
 UK           7475
 AU            767
 IN            676
 JP            593
 MY            570
 NZ            222
```

   (Hash subpartitioning of list partitions is not supported in Oracle9*i*) You decide the following: Move the CA customers from the CUST_NA (North America) to CUST_SA (South America). This requires splitting and merging. Also, the partition that now contains only US customers is to be named CUST_USA, and the other American partition will be CUST_AM.

   Merge the CUST_NA and CUST_SA into CUST_TMP in the DATA01 tablespace, as a temporary measure. You want to avoid rebuilding the global partitioned index.

2. Check index status. Hint: all relevant indexes start with CST.

3. Split the CUST_TMP into the desired CUST_USA and CUST_AM, placing them into tablespaces DATA03 and DATA04, respectively. “Forget” to maintain the global indexes.

4. Check index status. Note the partition key values, and the local index status.

5. The table SHIPPED_T appears to be too crowded in the last range partition, so you increase the number of subpartitions.

6. Because no storage specification was made, the subpartition ended up in the default tablespace, which is not the intention. Identify and move the subpartition to DATA04.

Practice 4-3: Exchange partition and table

1. Another season has passed, and it is time for the next rolling window operation of SHIPPED. However, an analyst wants to perform an in-depth analysis of the data in the SHP_Q2_2002 partition that you are about to discard, and asks that it be provided as a separate table.

   Create a suitable table, called OLD_SHIPPED in the USERS tablespace. Create an index on OLD_SHIPPED.PROD_ID.

2. Exchange SHP_Q2_2002 and OLD_SHIPPED, with the index.

3. What is the status of the involved data now, specifically:

3a. Which tablespace is the old SHIPPED data, now in the OLD_SHIPPED table, located?

____

3b. Can the old data be queried through the SHIPPED table?

____

3c. If you now drop the SHP_Q2_2002 partition, will there be any unexpected side effects??

____

3d. How might you get the old data out of the production tablespaces (DATAnn)?

____

4. Check the status of the indexes on both OLD_SHIPPED and SHIPPED.

Practice 5-1 Export and Import of Partition

This exercise demonstrates the use of Export and Import with partitioned tables and should be performed as user `sh`. Export the 1998 Q1 partition. Name the export dump file `sales_q1_1998.dmp` and make sure it resides in your home directory. Perform a query that accesses data in this partition, then truncate the `sales_q1_1998` partition. Use Import to restore the data.

1. Connect as user `sh` and confirm the SALES table partition names.
2. Perform the export. Make sure the dump file is written to your home directory.
3. Perform a query that accesses data in the SALES_Q1_1998 partition.
4. Truncate the data in the partition SALES_Q1_1998.
5. Verify that the data is gone.
6. Import the data back into the empty partition.
7. Repeat the same query executed previously to verify that the data has been restored.

Practice 5-2: Load a partition with SQL*Loader

This practice demonstrates how SQL*Loader works with partitioned tables. As user sh, truncate the SALES_Q1_1998 partition from the SALES table. The partition data will be loaded from the `sh_sales.dat file` located in `$ORACLE_HOME/demo/schema/sales_history` directory. Using the `sh_sales.ctl` control file as a model, create your own SQL*Loader control file in your home directory and reload the SALES_Q1_1998 partition.

1. Truncate the data in the SALES_Q1_1998 partition.

2. Verify that the partition is empty.

3. Make sure you are in your home directory. Copy the `sh_sales`.ctl file to `sales.ctl` and make the necessary edits.

4. Use SQL*Loader to load the data into the partition SALES_Q1_1998 partition.

5. Verify that the data has been successfully loaded.

Practice 5-3 Partitions in Transportable Tablespaces

This exercise demonstrates self-containment of partitioned tables in transportable tablespaces. Perform all steps of this exercise as `sysdba`. Any transportable tablespace candidate must be self-contained. Perform a self-containment check of the tablespace SAMPLE. Then move the SALES partition SALES_Q1_1998 to the USERS tablespace. Perform another self-containment check and observe the differences.

1. Check for self-containment, using the `dbms_tts.transport_set_check` procedure.
2. View any violations by querying the `TRANSPORT_SET_VIOLATIONS` table.
3. Give user `sh` unlimited quota on the USERS tablespace and move the SALES_Q1_PARTITION:
4. Rerun the self-containment check:
5. Check again for violations.

Practice 6-1 Rolling Window Operation

This exercise emphasizes the mechanics of performing rolling-window operations. Our attention will be focused on the fact table SALES in the SH schema. It has now become necessary to drop the oldest partition, SALES_q1_1998, and add a brand new SALES_q1_2001 partition. Perform the necessary steps to accomplish this task. Don't forget about index maintenance.

1. Connect as SH and query the partitions currently comprising the SALES table.

2. Drop the partition SALES_Q1_1998.

3. Add another partition SALES_Q1_2001 above the partition SALES_Q4_2000. Since that partition is bounded by MAXVALUE, you must split SALES_Q4_2000.

4. Check to see that the new SALES_Q1_2001 partition has been properly created.

5. The indexes for the fact table SALES must reflect the fact that you have dropped one partition and added another. Query the `USER_PART_INDEXES` view to determine the associated indexes for the table.

6. Identify the index partitions to be rebuilt. Select the index partition_name from the `USER_IND_PARTITIONS` view.

7. Rebuild the affected indexes.

Practice 6-2 Partitioned View to Partitioned Table Conversion

In this exercise, you will create a partition view and then complete the steps required to convert it to a partitioned table. As user `sh`, create three standard tables as select * from sales, partitions SALES_Q1_1999 through SALES_Q1_1999 inclusive. Create a partitioned view called SALES_PART_VIEW from the three newly created tables. Run the `$HOME/STUDENT/LABS/lab_06_02_view_to_table.sql` script to create an empty partitioned table called SALES_PART_TABLE. Exchange each partition with its corresponding table.

1. Create tables.

2. Create the partitioned view. Connect as `SYSDBA` and grant `create view` to the user sh to accomplish this.

3. Prepare for the migration by creating the partitioned table SALES_PART_TABLE. You can create it by running the script `$HOME/STUDENT/LABS/lab_06_02_view_to_table.sql`. Please inspect this script before you execute it. It will be empty in anticipation of the migrated data, so notice that a segment of two blocks is specified as an initial storage value to act as a placeholder.

4. Use the `EXCHANGE PARTITION` statement to migrate the tables to the corresponding partitions.

5. In the real world, you would then drop the original partitioned view and use the old view name to rename the new partitioned table so that the change would be transparent to the users.

Practice 6-3 A Very Mixed Table

In this exercise, you will execute the `lab_06_03_create_mix.sql` script located in the `$HOME/STUDENT/LABS` directory to create a table that will demonstrate partitioned table support of various data types, data organization, constraints, and so on. The table is called MIX and creates the following columns and datatypes:

NU – NUMBER
CH – CHAR
VC – VARCHAR
CL – CLOB
BL – BLOB
TS - TIMESTAMP

NU and VC are primary keys while CH and VC are unique. The table is range partitioned on the VC column. The MIX table uses tablespaces DATA01 through DATA04 and INDEX01 through INDEX04 for storage, both primary and overflow. Two local indexes are created, one on TS and another on VC and TS.

Spend a few moments and inspect the `lab_06_03_create_mix.sql` script. Pay special attention to the column datatypes, partitioning statements, storage parameters, constraints, and index creation.

1. Execute the script `$HOME/STUDENT/LABS/ lab_06_03_create_mix.sql`.
2. Check table and partition creation.
3. Look at the partitioned columns.
4. Look at the indexes associated with the MIX table.

B Solutions

Lesson 1 Practices

1. Prior to the introduction of the Oracle Partitioning option, manual partitioning was performed to address large table manageability. Can you list some difficulties that might be encountered when using manual partitioning?

   a. Optimizing queries and tuning can be more complex.

   b. Manageability becomes more complex as each manual table partition has its metadata definition.

   c. Primary keys and unique constraints are almost impossible to implement.

2. The Oracle Partitioning option offers many advantages to the database administrator when dealing with very large tables and indexes. Can you list some of these advantages?

   a. Using Oracle partitioning to divide tables and indexes into smaller partitions improves availability of data because if one partition is unavailable, other partitions can be used.

   b. Unavailable partitions do not affect queries or DML operations on other partitions that use the same table or index.

   c. Each partition can be managed individually, and can function independently of the other partitions, thus providing a structure that can be better tuned for availability and performance.

   d. Partitioning is transparent to existing applications as are standard DML statements run against partitioned tables.

   e. Partitions can be scanned, updated, inserted, or deleted in parallel, to improve performance.

   f. Partitions can be load-balanced across physical devices.

3. Please explain the concept and benefits of partition pruning.

   Depending on the SQL statement, the Oracle server can explicitly recognize partitions and subpartitions that need to be accessed, and the ones that can be eliminated. This elimination or optimization is called partition pruning. This can result in substantial improvements in query performance. Pruning is expressed using a range of partitions, and the relevant partitions for the query are all the partitions between the first and the last partition of that range.

4. Can you think of a situation where it would be beneficial to partition the index rather than the associated table?

> For OLTP applications in which the index is always used, it might be more useful to partition the index and not the table because the pruning at the index level is the primary obtainable performance gain.

5. List the four partitioning methods and briefly explain each one.

> a. Range Partitioning
> Range partitioning uses ranges of column values to map rows to partitions. Partitioning by range is well suited for historical databases.
>
> b. List Partitioning
> List Partitioning uses itemized lists of values of column values for each partition.
>
> c. Hash Partitioning
> Hash Partitioning uses a hashing algorithm to map rows to partitions. It is well suited if queries are made in parallel.
>
> d. Composite Partitioning (Hash subpartition of Range Partition)
> Composite Partitioning combines the advantages of range partitioning, (easier management), with the query benefits of more and smaller partitions by hash subpartitioning each range partition.

Practice 2 General Comments

Log in to the `DATAMGR` schema, using the password `DATAMGR` for these practices unless otherwise noted.

Because the creation commands are rather lengthy, it is recommended that you use scripts to make it easier to re-create the tables with variations.

Practice 2-1 Solution: Create Partitioned Tables of Each Type

1. Create a range-partitioned table. The table contains sales history data, and is partitioned by quarters. The table uses rolling window operations, where a new quarter is added, an old one is dropped.

   Table Structure: Name: SHIPPED. Columns: PROD_ID, CUST_ID, DATETIME, UNITS_SOLD, AMOUNT_SOLD, the data type of all columns is NUMBER, except the DATETIME which is TIMESTAMP WITH LOCAL TIME ZONE.

   Partition: Range partition on DATETIME. The partitions for each quarter are to be named SHP_Qn_yyyy where "n" is 1 to 4 and "yyyy" is the year. The first partition is SHP_Q1_2002; the last one is SHP_Q2_2003, 6 partitions. Each partition should contain the appropriate rows based on the DATETIME value.

   Storage: The partitions for year 2002 go to tablespace DATA02, the year 2003 into DATA03, and so on. Initially, you are loading bulk data so all partitions must have PCTFREE 5, except the last one which needs PCTFREE 20. Other storage attributes can be default values.

   Note: To specify a TIMESTAMP literal, the syntax is
   `TIMESTAMP 'yyyy-mm-dd hh:mm:ss.ff +hh:mm'`
   This syntax is fixed, invariant of the NLS settings.

```
SQL> CREATE TABLE shipped
     ( prod_id   NUMBER
     , cust_id   NUMBER
     , datetime       TIMESTAMP WITH LOCAL TIME ZONE
     , quantity       NUMBER
     , amount    NUMBER(10,2)
     )  PCTFREE 5
      PARTITION BY RANGE (datetime)
      ( PARTITION SHP_Q1_2002 VALUES LESS THAN
            (TIMESTAMP '2002-04-01 00:00:00.00 +00:00')
        TABLESPACE data02
      , PARTITION SHP_Q2_2002 VALUES LESS THAN
            (TIMESTAMP '2002-07-01 00:00:00.00 +00:00')
        TABLESPACE data02
      , PARTITION SHP_Q3_2002 VALUES LESS THAN
            (TIMESTAMP '2002-10-01 00:00:00.00 +00:00')
        TABLESPACE data02
      , PARTITION SHP_Q4_2002 VALUES LESS THAN
            (TIMESTAMP '2003-01-01 00:00:00.00 +00:00')
        TABLESPACE data02
      , PARTITION SHP_Q1_2003 VALUES LESS THAN
```

```
 22         (TIMESTAMP '2003-04-01 00:00:00.00 +00:00')
 23      TABLESPACE data03
 24    , PARTITION SHP_Q2_2003 VALUES LESS THAN
 25         (TIMESTAMP '2003-07-01 00:00:00.00 +00:00')
 26      TABLESPACE data03 PCTFREE 20
 27    )
 28    ;

Table created.
```

2. Examine the data dictionary views to verify that the partition definitions and storage attributes are defined correctly.

```
SQL> SELECT TABLE_NAME, PARTITION_NAME, HIGH_VALUE,
  2    TABLESPACE_NAME, PCT_FREE
  3    FROM USER_TAB_PARTITIONS ;

TABLE_NAME Part.Name          HIGH_VALUE      TABLESPACE   PCT_FREE
---------- ------------------ --------------- ---------- ----------
SHIPPED    SHP_Q1_2002        'TIMESTAMP'2002 DATA02              5
                              -04-01 00:00:00
                              .000000000+00:0
                              0 '
SHIPPED    SHP_Q2_2002        TIMESTAMP'      DATA02              5
SHIPPED    SHP_Q3_2002        TIMESTAMP'      DATA02              5
SHIPPED    SHP_Q4_2002        TIMESTAMP'      DATA02              5
SHIPPED    SHP_Q1_2003        TIMESTAMP'      DATA03              5
SHIPPED    SHP_Q2_2003        TIMESTAMP'      DATA03             20
```

The printout is slightly reformatted to fit. The HIGH_VALUE column's value is only partially shown after the first record.

3. Create a list-partitioned table. The table contains customer addresses that include a country code, and should be partitioned by continent. This is in anticipation of the need for extensive data manipulation, which will occur region-by-region.

   Table Structure: Name: CUSTS. Columns and datatype: CUST_ID NUMBER, FIRSTNAME VARCHAR2(20), LASTNAME VARCHAR2(40), ADDRESS VARCHAR2(20), COUNTRY CHAR(2).

   Partition: List partition on `COUNTRY`.

Partition name (Region)	Partition Key (Country codes)	Comment
CUST_AM	US, CA	North America
CUST_SA	AR, BR	South America
CUST_EU	DE, FR, UK, DK, ES, IE, NL	Europe
CUST_XX	AU, IN, JP, MY, NZ	Others

   NULL is to be an allowed value.

   Storage: All of the partitions are to be stored in `DATA04`. Other storage attributes can be default values.

```
SQL> CREATE TABLE custs
```

```
  2    ( cust_id      NUMBER
  3    , firstname         VARCHAR2(20)
  4    , lastname          VARCHAR2(40)
  5    , address      VARCHAR2(40)
  6    , country      CHAR(2)
  7    ) TABLESPACE data04
  8     PARTITION BY LIST ( country )
  9    ( PARTITION cust_na VALUES ( 'US', 'CA' )
 10    , PARTITION cust_sa VALUES ( 'AR', 'BR' )
 11    , PARTITION cust_eu VALUES ( 'DE', 'FR', 'UK', 'DK',
 12                         'ES', 'IE', 'NL' )
 13    , PARTITION cust_xx VALUES ( 'AU', 'IN', 'JP', 'MY', 'NZ',
 14                          NULL )
 15    ) ;

Table created.
```

4. Examine the data dictionary views to verify the partition key values.

```
TABLE_NAME   Part.Name          HIGH_VALUE             TABLESPACE
------------ ------------------ ---------------------- ----------
CUSTS        CUST_NA            'US', 'CA'             DATA04
CUSTS        CUST_SA            'AR', 'BR'             DATA04
CUSTS        CUST_EU            'DE', 'FR', 'UK', 'DK' DATA04
                                , 'ES', 'IE', 'NL'
CUSTS        CUST_XX            'AU', 'IN', 'JP', 'MY' DATA04
                                , 'NZ', NULL
```

5. Create a hash-partitioned table. The hashing is required to ease management operation with the table.

   Table Structure: Name: TEST_RESULT. Columns: TEST_ID NUMBER, BATCH_NO NUMBER, RESULT_A VARCHAR2(4000), RESULT_B VARCHAR2(4000).

   Partition: HASH partion on TEST_ID into 8 partitions. The partition names are irrelevant.

   Storage: The partitions are to be evenly spread in tablespaces DATA01, DATA02, and DATA03. Partition segments must have the initial segment size of 200K.

```
SQL> CREATE TABLE test_result
  2      ( test_id         NUMBER
  3      , batch_no        NUMBER
  4      , result_a        VARCHAR2(4000)
  5      , result_b        VARCHAR2(4000)
  6      ) STORAGE ( INITIAL 200K )
  7    PARTITION BY HASH ( TEST_ID )
  8      PARTITIONS 8
  9      STORE IN ( data01, data02, data03 )
 10    ;
```

6. Examine the data dictionary views to verify that the consumed storage amount is correct. **Note**: This will differ from the specified initial extent size, if the tablespace is locally managed with Automatic or Uniform Extent Size, or has had Minimum Extent defined.

```
SQL> SELECT SEGMENT_NAME, PARTITION_NAME, BYTES
  2    FROM USER_SEGMENTS
  3    WHERE SEGMENT_NAME='TEST_RESULT' ;

SEGMENT_NAME       PARTITION_NAME          BYTES
------------------ ----------------- ----------
TEST_RESULT        SYS_P786              262144
TEST_RESULT        SYS_P787              262144
TEST_RESULT        SYS_P788              262144
TEST_RESULT        SYS_P789              262144
TEST_RESULT        SYS_P790              262144
TEST_RESULT        SYS_P791              262144
TEST_RESULT        SYS_P792              262144
TEST_RESULT        SYS_P793              262144
```

The system generated names may differ on your output

7. Create a range-hash composite partitioned table. The SHIPPED table from step 1 above needs subpartitioning. Because you want to use the DBMS_REDEFENITION package to migrate, you will create the new table structure as a first step. (The DBMS_REDEFINITION, which allows for table restructuring while the data, including updates, remains available for users, will not be covered in this course.)

   Table Structure: Table name is SHIPPED_T. Table structure is the same as for SHIPPED. You can choose to use CREATE TABLE … SELECT AS … WHERE ROWNUM<1 to copy the structure.

   Partition: Range partition is the same as for the SHIPPED table.. Subpartition on columns CUST_ID and PROD_ID. Four subpartitions per partition. Names are irrelevant. Allow for large changes in the DATETIME value.

   Storage: Subpartition segments to be stored in DATA02 for the year 2002, and DATA03 for the year 2003, except for the last partition (SHP_Q2_2003), which needs the subpartitions stored in DATA01 and DATA04.

```
SQL> CREATE TABLE shipped_t
  2    PCTFREE 5
  3    PARTITION BY RANGE (datetime)
  4      SUBPARTITION BY HASH ( cust_id, prod_id )
  5        SUBPARTITIONS 4 STORE IN ( data02 )
  6    ( PARTITION SHP_Q1_2002 VALUES LESS THAN
  7           (TIMESTAMP '2002-04-01 00:00:00.00 +00:00')
  8    , PARTITION SHP_Q2_2002 VALUES LESS THAN
  9           (TIMESTAMP '2002-07-01 00:00:00.00 +00:00')
 10    , PARTITION SHP_Q3_2002 VALUES LESS THAN
 11           (TIMESTAMP '2002-10-01 00:00:00.00 +00:00')
 12    , PARTITION SHP_Q4_2002 VALUES LESS THAN
 13           (TIMESTAMP '2003-01-01 00:00:00.00 +00:00')
 14    , PARTITION SHP_Q1_2003 VALUES LESS THAN
 15           (TIMESTAMP '2003-04-01 00:00:00.00 +00:00')
 16        SUBPARTITIONS 4 STORE IN ( data03 )
 17    , PARTITION SHP_Q2_2003 VALUES LESS THAN
 18           (TIMESTAMP '2003-07-01 00:00:00.00 +00:00')
```

```
 19        SUBPARTITIONS 4 STORE IN ( data01, data04 )
 20    )
 21    ENABLE ROW MOVEMENT
 22    AS SELECT * FROM shipped
 23      WHERE ROWNUM<1
 24    ;

Table created.
```

Note the `ROW MOVEMENT` *clause on line 18; “allow for large changes in the DATETIME value”*

8. Use the Data dictionary view to determine the tablespace in which the subpartitions are stored.

```
SQL> SELECT TABLE_NAME, PARTITION_NAME,
  2    SUBPARTITION_NAME, TABLESPACE_NAME
  3    FROM USER_TAB_SUBPARTITIONS
  4    WHERE TABLE_NAME='SHIPPED_T'
  5    ORDER BY SUBPARTITION_NAME ;

TABLE_NAME Part.Name          Sub.P.Name         TABLESPACE
---------- ------------------ ------------------ ----------
SHIPPED_T  SHP_Q1_2002        SYS_SUBP1258       DATA02
SHIPPED_T  SHP_Q1_2002        SYS_SUBP1259       DATA02
SHIPPED_T  SHP_Q1_2002        SYS_SUBP1260       DATA02
SHIPPED_T  SHP_Q1_2002        SYS_SUBP1261       DATA02
SHIPPED_T  SHP_Q2_2002        SYS_SUBP1262       DATA02
SHIPPED_T  SHP_Q2_2002        SYS_SUBP1263       DATA02
SHIPPED_T  SHP_Q2_2002        SYS_SUBP1264       DATA02
SHIPPED_T  SHP_Q2_2002        SYS_SUBP1265       DATA02
SHIPPED_T  SHP_Q3_2002        SYS_SUBP1266       DATA02
SHIPPED_T  SHP_Q3_2002        SYS_SUBP1267       DATA02
SHIPPED_T  SHP_Q3_2002        SYS_SUBP1268       DATA02
SHIPPED_T  SHP_Q3_2002        SYS_SUBP1269       DATA02
SHIPPED_T  SHP_Q4_2002        SYS_SUBP1270       DATA02
SHIPPED_T  SHP_Q4_2002        SYS_SUBP1271       DATA02
SHIPPED_T  SHP_Q4_2002        SYS_SUBP1272       DATA02
SHIPPED_T  SHP_Q4_2002        SYS_SUBP1273       DATA02
SHIPPED_T  SHP_Q1_2003        SYS_SUBP1274       DATA03
SHIPPED_T  SHP_Q1_2003        SYS_SUBP1275       DATA03
SHIPPED_T  SHP_Q1_2003        SYS_SUBP1276       DATA03
SHIPPED_T  SHP_Q1_2003        SYS_SUBP1277       DATA03
SHIPPED_T  SHP_Q2_2003        SYS_SUBP1278       DATA01
SHIPPED_T  SHP_Q2_2003        SYS_SUBP1279       DATA04
SHIPPED_T  SHP_Q2_2003        SYS_SUBP1280       DATA01
SHIPPED_T  SHP_Q2_2003        SYS_SUBP1281       DATA04

24 rows selected.
```

```
SQL> SELECT SEGMENT_NAME,PARTITION_NAME,
  2     SEGMENT_TYPE,TABLESPACE_NAME
  3     FROM USER_SEGMENTS
```

```
  4     WHERE SEGMENT_NAME='SHIPPED_T'
  5    ORDER BY PARTITION_NAME ;

SEGMENT_NAME       Part.Name          SEGMENT_TYPE       TABLESPACE
------------------ ------------------ ------------------ ----------
SHIPPED_T          SYS_SUBP1258       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1259       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1260       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1261       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1262       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1263       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1264       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1265       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1266       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1267       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1268       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1269       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1270       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1271       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1272       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1273       TABLE SUBPARTITION DATA02
SHIPPED_T          SYS_SUBP1274       TABLE SUBPARTITION DATA03
SHIPPED_T          SYS_SUBP1275       TABLE SUBPARTITION DATA03
SHIPPED_T          SYS_SUBP1276       TABLE SUBPARTITION DATA03
SHIPPED_T          SYS_SUBP1277       TABLE SUBPARTITION DATA03
SHIPPED_T          SYS_SUBP1278       TABLE SUBPARTITION DATA01
SHIPPED_T          SYS_SUBP1279       TABLE SUBPARTITION DATA04
SHIPPED_T          SYS_SUBP1280       TABLE SUBPARTITION DATA01
SHIPPED_T          SYS_SUBP1281       TABLE SUBPARTITION DATA04

24 rows selected.
```

Either data dictionary table gives the result. Note that the subpartition name is listed in the PARTITION_NAME column.

9. The TEST_RESULT table needs to be re-created. The result text is too large for VARCHAR2(4000) and rather than add another RESULT_3 and so on, a CLOB will be used. Fortunately, the data can be reloaded so that you can drop and recreate the table.

   <u>Table structure:</u> As above, but instead of RESULT_1 and RESULT_2, use a column named RESULTS of type CLOB.

   <u>Partition:</u> No change to the table partition specification for TEST_RESULT from earlier.

   <u>Storage:</u> The table partitions are all to be stored in DATA04 without specifying any initial sizing. The CLOB partitions are to be spread in DATA01, DATA02 and DATA03.

```
SQL> DROP TABLE test_result ;

Table dropped.
```

```
SQL> CREATE TABLE test_result
  2      ( test_id          NUMBER
  3      , batch_no         NUMBER
  4      , results          CLOB
  5      ) TABLESPACE data04
  6    PARTITION BY HASH ( TEST_ID )
  7      ( PARTITION h_1
  8        LOB ( results ) STORE AS hl_1 ( TABLESPACE data01 )
  9      , PARTITION h_2
 10        LOB ( results ) STORE AS hl_2 ( TABLESPACE data02 )
 11      , PARTITION h_3
 12        LOB ( results ) STORE AS hl_3 ( TABLESPACE data03 )
 13      , PARTITION h_4
 14        LOB ( results ) STORE AS hl_4 ( TABLESPACE data01 )
 15      , PARTITION h_5
 16        LOB ( results ) STORE AS hl_5 ( TABLESPACE data02 )
 17      , PARTITION h_6
 18        LOB ( results ) STORE AS hl_6 ( TABLESPACE data03 )
 19      , PARTITION h_7
 20        LOB ( results ) STORE AS hl_7 ( TABLESPACE data01 )
 21      , PARTITION h_8
 22        LOB ( results ) STORE AS hl_8 ( TABLESPACE data02 )
 23      )  ;

Table created.
```

You cannot specify LOB storage if specifying hash partitions by quantity, instead of named.

10. Check the tablespace allocation. Hint: All partition names of this table start with ‘H’.

```
SQL> SELECT SEGMENT_NAME, PARTITION_NAME, SEGMENT_TYPE, TABLESPACE_NAME
  2    FROM USER_SEGMENTS
  3     WHERE PARTITION_NAME LIKE 'H%' ;

SEGMENT_NAME               PARTIT SEGMENT_TYPE       TABLESPACE
-------------------------- ------ ------------------ ----------
TEST_RESULT                H_1    TABLE PARTITION    DATA04
TEST_RESULT                H_2    TABLE PARTITION    DATA04
TEST_RESULT                H_3    TABLE PARTITION    DATA04
TEST_RESULT                H_4    TABLE PARTITION    DATA04
TEST_RESULT                H_5    TABLE PARTITION    DATA04
TEST_RESULT                H_6    TABLE PARTITION    DATA04
TEST_RESULT                H_7    TABLE PARTITION    DATA04
TEST_RESULT                H_8    TABLE PARTITION    DATA01
SYS_LOB0000008904C00003$$ HL_1   LOB PARTITION      DATA01
SYS_LOB0000008904C00003$$ HL_2   LOB PARTITION      DATA02
SYS_LOB0000008904C00003$$ HL_3   LOB PARTITION      DATA03
SYS_LOB0000008904C00003$$ HL_4   LOB PARTITION      DATA01
SYS_LOB0000008904C00003$$ HL_5   LOB PARTITION      DATA02
SYS_LOB0000008904C00003$$ HL_6   LOB PARTITION      DATA03
SYS_LOB0000008904C00003$$ HL_7   LOB PARTITION      DATA01
SYS_LOB0000008904C00003$$ HL_8   LOB PARTITION      DATA02

16 rows selected.
```

Note: At the end of this practice, you should have four tables in the DATAMGR schema, that will be used in subsequent practices.

Practice 2-2 Use the data dictionary to verify the partition structure

1. Create a “plain vanilla” table. Name: VANILLA. Columns: DATA of NUMBER and TEXT of VARCHAR2(20). No partitioning. Place in tablespace USERS.

```
SQL> CREATE TABLE vanilla
  2   ( DATA NUMBER, TEXT VARCHAR2(20) )
  3    TABLESPACE users ;

Table created.
```

2. Display the table name, partition type, and row movement status of the tables created so far. The nonpartitioned table should be identified as such.

```
SQL> SELECT TABLE_NAME, PARTITIONED, ROW_MOVEMENT
  2    FROM USER_TABLES ;

TABLE_NAME   PAR ROW_MOVE
------------ --- --------
CUSTS        YES DISABLED
SHIPPED      YES DISABLED
SHIPPED_T    YES ENABLED
TEST_RESULT  YES DISABLED
VANILLA      NO  DISABLED
```

```
SQL> SELECT TABLE_NAME, PARTITIONING_TYPE, SUBPARTITIONING_TYPE
  2    FROM USER_PART_TABLES ;

TABLE_NAME   PARTITI SUBPART
------------ ------- -------
CUSTS        LIST    NONE
SHIPPED      RANGE   NONE
SHIPPED_T    RANGE   HASH
TEST_RESULT  HASH    NONE
```

```
SQL> REM The same information in one query
SQL> SELECT TABLE_NAME, PARTITIONED,
  2         ROW_MOVEMENT, NVL(PARTITIONING_TYPE,'N/A') PART_TYPE,
  3         NVL(SUBPARTITIONING_TYPE,'N/A') SUBPART_TYPE
  4    FROM USER_PART_TABLES NATURAL RIGHT JOIN USER_TABLES ;

TABLE_NAME   PAR ROW_MOVE PART_TY SUBPART
------------ --- -------- ------- -------
VANILLA      NO  DISABLED N/A     N/A
SHIPPED_T    YES ENABLED  RANGE   HASH
CUSTS        YES DISABLED LIST    NONE
TEST_RESULT  YES DISABLED HASH    NONE
SHIPPED      YES DISABLED RANGE   NONE
```

3. Display the table names that have some table partition in tablespace DATA03. Ignore LOB segments.

```
SQL> SELECT UNIQUE SEGMENT_NAME TABLE_PARTS
       FROM USER_SEGMENTS
       WHERE SEGMENT_TYPE IN
        ('TABLE PARTITION','TABLE SUBPARTITION')
       AND TABLESPACE_NAME='DATA03'
       ;

TABLE_PARTS
--------------------------------------------------------
SHIPPED
SHIPPED_T
```

4. Display the partition columns used for SHIPPED and SHIPPED_T.

```
SQL> SELECT NAME TABLE_NAME, 'PART' PART, COLUMN_NAME,
         COLUMN_POSITION "COL.POS."
       FROM USER_PART_KEY_COLUMNS
       WHERE NAME IN ('SHIPPED','SHIPPED_T')
     UNION ALL
     SELECT NAME TABLE_NAME, 'SUBP' PART, COLUMN_NAME,
         COLUMN_POSITION "COL.POS."
       FROM USER_SUBPART_KEY_COLUMNS
       WHERE NAME IN ('SHIPPED','SHIPPED_T')
       ;

TABLE_NAME   PART COLUMN_NAME       COL.POS.
------------ ---- --------------- ----------
SHIPPED      PART DATETIME                 1
SHIPPED_T    PART DATETIME                 1
SHIPPED_T    SUBP CUST_ID                  1
SHIPPED_T    SUBP PROD_ID                  2
```

5. Log in to the SH sample schema, password SH.

```
CONNECT SH/SH

Connected.
```

6. Find which tables are partitioned. Determine how many rows are contained in one of the partitioned tables, and in one of the table's partitions.

```
SQL> REM The same query as from step 2
SQL> SELECT TABLE_NAME, PARTITIONED,
  2         ROW_MOVEMENT, NVL(PARTITIONING_TYPE,'N/A') PART_TYPE,
  3         NVL(SUBPARTITIONING_TYPE,'N/A') SUBPART_TYPE
  4    FROM USER_PART_TABLES NATURAL RIGHT JOIN USER_TABLES ;

TABLE_NAME              PAR ROW_MOVE PART_TY SUBPART
----------------------- --- -------- ------- -------
TIMES                   NO  DISABLED N/A     N/A
CHANNELS                NO  DISABLED N/A     N/A
PROMOTIONS              NO  DISABLED N/A     N/A
COUNTRIES               NO  DISABLED N/A     N/A
CUSTOMERS               NO  DISABLED N/A     N/A
PRODUCTS                NO  DISABLED N/A     N/A
CAL_MONTH_SALES_MV      NO  DISABLED N/A     N/A
...
SALES_TRANSACTIONS_EXT  NO  DISABLED N/A     N/A
COSTS                   YES DISABLED RANGE   NONE
SALES                   YES DISABLED RANGE   NONE
```

```
SQL> SELECT TABLE_NAME, PARTITION_NAME
  2    FROM USER_TAB_PARTITIONS ;

TABLE_NAME                     PARTITION_NAME
------------------------------ ---------------------
SALES                          SALES_Q1_1998
SALES                          SALES_Q2_1998
SALES                          SALES_Q3_1998
SALES                          SALES_Q4_1998
SALES                          SALES_Q1_1999
...
24 rows selected.
```

```
SQL> SELECT COUNT(*) FROM SALES ;

  COUNT(*)
----------
   1016271
```

```
SQL> SELECT COUNT(*) FROM SALES PARTITION ( SALES_Q1_2000 ) ;

  COUNT(*)
----------
    104544
```

7. Log in again to the DATAMGR schema.

```
Connect DATAMGR/DATAMGR

Connected.
```

Note: At the end of this practice, you should have 5 tables in the DATAMGR schema.

Practice 2-3 See row placement in partitions

1. Insert a few rows into the CUSTS table. For example, 3 rows with a customer residing in the countries US, CA, and DE, respectively.

```
SQL> INSERT INTO custs VALUES
  2   ( 1, 'Alpha', 'Primus',  'First Street', 'CA' ) ;
SQL> INSERT INTO custs VALUES
  2   ( 2, 'Beta',  'Secundus','Zweite Strasse', 'DE' ) ;
SQL> INSERT INTO custs VALUES
  2   ( 3, 'Gamma', 'Tertius', 'Troisieme Street', 'CA' ) ;
SQL> COMMIT ;

Commit complete.
```

2. Select the block number of each row's rowid. The function DBMS_ROWID.ROWID_BLOCK_NUMBER(rowid) accomplishes this.

```
SQL> SELECT DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) BLOCK,
  2         cust_id, country
  3    FROM custs ;

     BLOCK    CUST_ID CO
---------- ---------- --
        18          1 CA
        18          3 CA
        34          2 DE
```

Note the grouping of the rows by the partition. (Your block numbers may vary)

3. Populate the table with data from the SH schema's CUSTOMER table, using

```
INSERT INTO custs
  SELECT cust_id, cust_first_name, cust_last_name,
         cust_street_address,country_id
  FROM sh.customers ;
```

The insert should fail. Select an appropriate subset of the data so the insert succeeds, and commit the insert.

```
SQL> INSERT INTO custs
  2  SELECT cust_id, cust_first_name, cust_last_name,
  3         cust_street_address, country_id
  4    FROM sh.customers ;
INSERT INTO custs
            *
ERROR at line 1:
ORA-14400: inserted partition key does not map to any partition
```

```
SQL> INSERT INTO custs
  2  SELECT cust_id, cust_first_name, cust_last_name,
  3         cust_street_address, country_id
  4    FROM sh.customers
  5    WHERE COUNTRY_ID IN ('US', 'CA', 'AR', 'BR',
  6      'DE', 'FR', 'UK', 'DK', 'ES', 'IE', 'NL', 'PL', 'TR',
  7      'AU', 'IN', 'JP', 'MY', 'NZ', NULL ) ;

49873 rows created.

SQL> Commit ;

Commit complete.
```

4. As SYSTEM/MANAGER, examine the DBA_EXTENTS view to determine which blocks belong to a particular partition segment of the CUSTS table. Compare this to the DBMS_ROWID.ROWID_BLOCK_NUMBER(ROWID) returned from a query of some customers, for example, all customers in CUSTS with CUST_ID less than 200, and verify that rows are placed in the right partition.

```
SQL> CONNECT SYSTEM/MANAGER
Connected.
SQL> SELECT DBA_EXTENTS.PARTITION_NAME PARTITION, custs.country,
  2        DBMS_ROWID.ROWID_BLOCK_NUMBER(custs.ROWID) BLOCK, custs.cus
  3   FROM DBA_EXTENTS CROSS JOIN datamgr.custs
  4   WHERE cust_id < 200
  5   AND   DBA_EXTENTS.OWNER='DATAMGR'
  6   AND   DBA_EXTENTS.SEGMENT_NAME='CUSTS'
  7   AND   DBMS_ROWID.ROWID_BLOCK_NUMBER(CUSTS.ROWID)
  8            BETWEEN BLOCK_ID AND BLOCK_ID+BLOCKS-1 ;

PARTITION                      CO      BLOCK    CUST_ID
------------------------------ -- ---------- ----------
CUST_NA                        CA         18          1
CUST_NA                        CA         18          3
CUST_NA                        US         18         20
CUST_NA                        US         18         30
CUST_NA                        US         18         40
CUST_NA                        US         18         70
CUST_NA                        US         18         90
CUST_NA                        US         18        110
CUST_NA                        US         18        190
CUST_EU                        DE         50          2
CUST_EU                        UK         50         10
CUST_EU                        FR         50         50
CUST_EU                        UK         50         60
CUST_EU                        ES         50         80
CUST_EU                        ES         50        100
CUST_EU                        NL         50        120
CUST_EU                        ES         50        130
CUST_EU                        ES         50        140
CUST_EU                        ES         50        150
```

```
CUST_EU                              ES           50          160
CUST_EU                              NL           50          170
CUST_EU                              DE           50          180
22 rows selected.
```

You can of course take any other “sample” of rows. Your block number may vary

Log back in to the DATAMGR schema after completing this exercise.

```
SQL> CONNECT DATAMGR/DATAMGR
Connected.
```

Practice 2-4 Verify Partitioning Pruning Takes Place

1. Create the table PLAN_TABLE by executing the utlxplan standard script. **Note**: This is located in ORACLE_HOME/rdbms/admin.

```
@?/rdbms/admin/utlxplan

Table created.
```

2. Use the EXPLAIN PLAN statement to find the execution plan for:

   - a query of the whole of the SHIPPED table
   - a query of named partition of the SHIPPED table
   - a query of a range of values in the DATETIME column of the SHIPPED table, for example 1-JUN-2001 to 31-AUG-2001.
   - A query of some countries from the CUSTS table, using a IN list, for example COUNTRY IN ('DE', 'FR', 'UK')

   Use the SET STATEMENT_ID clause to distinguish your separate plans, or TRUNCATE the PLAN_TABLE between each execution plan.

```
SQL> EXPLAIN PLAN SET STATEMENT_ID='1:FULL' FOR
  2    SELECT * FROM shipped ;

Explained.
```

```
SQL> EXPLAIN PLAN SET STATEMENT_ID='2:PART' FOR
  2    SELECT * FROM shipped PARTITION ( SHP_Q3_2002 ) ;

Explained.
```

```
SQL> EXPLAIN PLAN SET STATEMENT_ID='3:RANGE' FOR
  2    SELECT * FROM shipped
  3      WHERE datetime BETWEEN '1-JUN-2002' AND '31-AUG-2002'  ;

Explained.
```

```
SQL> EXPLAIN PLAN SET STATEMENT_ID='4:IN-list' FOR
  2    SELECT * FROM custs
  3      WHERE country IN ('DE','FR', 'UK')  ;

Explained.
```

3. View the execution plans. You need only view the columns STATEMENT_ID, OPERATION, PARTITION_START, and PARTITION_STOP from the table PLAN_TABLE.

```
SQL> SELECT STATEMENT_ID, OPERATION,
  2     PARTITION_START||':'||PARTITION_STOP PARTITIONS
  3  FROM PLAN_TABLE
  4  ORDER BY STATEMENT_ID, ID ;

STATEMENT_ OPERATION              PARTITIONS
---------- ---------------------- ------------------------------
1:FULL     SELECT STATEMENT       :
1:FULL     PARTITION RANGE        1:6
1:FULL     TABLE ACCESS           1:6
2:PART     SELECT STATEMENT       :
2:PART     TABLE ACCESS           3:3
3:RANGE    SELECT STATEMENT       :
3:RANGE    FILTER                 :
3:RANGE    PARTITION RANGE        KEY:KEY
3:RANGE    TABLE ACCESS           KEY:KEY
4:IN-list  SELECT STATEMENT       :
4:IN-list  PARTITION LIST         KEY(INLIST):KEY(INLIST)
4:IN-list  TABLE ACCESS           KEY(INLIST):KEY(INLIST)
```

4. Populate the SHIPPED table with sample data from the SH schema's SALES table.

```
INSERT INTO shipped
  SELECT prod_id, cust_id, (3*365)+time_id,
         quantity_sold, amount_sold
  FROM sh.sales
  WHERE time_id between '3-JAN-1999' and '30-JUN-2000'
  and   cust_id < 10000
  and   prod_id < 5000  ;
```

The command is available in `$HOME/STUDENT/LABS/lab_02_04_populate_shipped.sql`.

```
SQL> INSERT INTO shipped
  2    SELECT prod_id, cust_id, (3*365)+time_id,
  3           quantity_sold, amount_sold
  4    FROM sh.sales
  5    WHERE time_id between '3-JAN-1999' and '30-JUN-2000'
  6    and   cust_id < 10000
  7    and   prod_id < 5000  ;

47575 rows created.

SQL> Commit ;

Commit complete.
```

5. Repeat the execution plan from above for the SHIPPED table. Use a suffix on the STATEMENT_ID to distinguish the first round from this round. Check the execution plans now.

```
SQL> EXPLAIN PLAN SET STATEMENT_ID='1:FULL-2' FOR
  2    SELECT * FROM shipped ;

Explained.

   :
```

```
STATEMENT_ OPERATION              PARTITIONS
---------- ---------------------- ------------------------------
1:FULL     SELECT STATEMENT       :
1:FULL     PARTITION RANGE        1:6
1:FULL     TABLE ACCESS           1:6
1:FULL-2   SELECT STATEMENT       :
1:FULL-2   PARTITION RANGE        1:6
1:FULL-2   TABLE ACCESS           1:6
2:PART     SELECT STATEMENT       :
2:PART     TABLE ACCESS           3:3
2:PART-2   SELECT STATEMENT       :
2:PART-2   TABLE ACCESS           3:3
3:RANGE    SELECT STATEMENT       :
3:RANGE    FILTER                 :
3:RANGE    PARTITION RANGE        KEY:KEY
3:RANGE    TABLE ACCESS           KEY:KEY
3:RANGE-2  SELECT STATEMENT       :
3:RANGE-2  FILTER                 :
3:RANGE-2  PARTITION RANGE        KEY:KEY
3:RANGE-2  TABLE ACCESS           KEY:KEY
   :
```

Note that the pruning takes place without any statistics or data being present.

6. OPTIONAL

Repeat step 4 from the practice 2-3 to verify correct placement of rows in the SHIPPED table.

```
SQL> REM Must be DBA priviledged to see DBA_EXTENTS
SQL> SELECT DBA_EXTENTS.PARTITION_NAME PARTITION, shipped.datetime,
  2        DBMS_ROWID.ROWID_BLOCK_NUMBER(datamgr.shipped.ROWID) BLOCK,
  3        shipped.cust_id,shipped.prod_id
  4   FROM DBA_EXTENTS CROSS JOIN datamgr.shipped
  5   WHERE cust_id =100 and shipped.prod_id<300
  6   AND   DBA_EXTENTS.OWNER='DATAMGR'
  7   AND   DBA_EXTENTS.SEGMENT_NAME='SHIPPED'
  8   AND   DBMS_ROWID.ROWID_BLOCK_NUMBER(datamgr.shipped.ROWID)
  9             BETWEEN BLOCK_ID AND BLOCK_ID+BLOCKS-1 ;

PARTITION    DATETIME                BLOCK    CUST_ID    PROD_ID
------------ --------------- ---------- ---------- ----------
SHP_Q1_2002  08-JAN-02 12.00         21        100        285
SHP_Q1_2002  15-JAN-02 12.00         25        100        285
```

```
SHP_Q1_2002  22-JAN-02 12.00          29         100         285
    :                                  :
SHP_Q2_2002  09-APR-02 12.00          39         100         255
SHP_Q2_2002  16-APR-02 12.00          43         100         255
SHP_Q2_2002  23-APR-02 12.00          48         100         255
SHP_Q2_2002  11-JUN-02 12.00         256         100          25
SHP_Q2_2002  28-APR-03 12.00         250         100          80
SHP_Q2_2002  16-JUN-02 12.00         259         100          25
SHP_Q2_2002  21-JUN-02 12.00         262         100          25
SHP_Q2_2002  26-JUN-02 12.00         265         100          25
SHP_Q3_2002  08-JUL-02 12.00          53         100         240
SHP_Q3_2002  15-JUL-02 12.00          57         100         240
    :                                  :
48 rows selected.
```

Remember to connect back into the DATAMGR schema.

Practice 3-1 Create most types of partitioned index

1. Create a normal, nonpartitioned index on the partitioned SHIPPED table.

   Index: Name SHP_NP_CI. Index the column CUST_ID.

   Storage: Default

```
SQL> CREATE INDEX shp_np_ci
  2    ON shipped ( cust_id ) ;

Index created.
```

2. Create a global partitioned index on CUSTS table.

   Index: Name CST_GL_LFN. Index on the columns LASTNAME, FIRSTNAME.

   Partition: Range partition. Try first the FIRSTNAME column as the partition key. Name the partitions: C_G_1, C_G_2, C_G_3 with end values ‘A’, ‘G’ and MAXVALUE, respectively.

   Storage: Use tablespaces INDX01, INDX02, and INDX03, one for each.

   Why will the index creation fail?

```
SQL> CREATE INDEX cst_gl_lfn
  2   ON custs(lastname, firstname)
  3    GLOBAL
  4    PARTITION BY RANGE (firstname)
  5   ( PARTITION c_g_1 VALUES LESS THAN  ('H')
  6   TABLESPACE indx01
  7   , PARTITION c_g_2 VALUES LESS THAN  ('Q')
  8   TABLESPACE indx02
  9   , PARTITION c_g_3 VALUES LESS THAN  (MAXVALUE)
 10   TABLESPACE indx03
 11   )  ;
  PARTITION BY RANGE (firstname)
                               *
ERROR at line 4:
ORA-14038: GLOBAL partitioned index must be prefixed
```

Non-prefixed Partitioned Global Indexes are not supported.

Do it with the LASTNAME column as the partition key, but otherwise use the same definition.

```
SQL> CREATE INDEX cst_gl_lfn
  2    ON custs(lastname, firstname)
  3     GLOBAL
  4     PARTITION BY RANGE (lastname)
  5    ( PARTITION c_g_1 VALUES LESS THAN  ('H')
  6    TABLESPACE indx01
  7    , PARTITION c_g_2 VALUES LESS THAN  ('Q')
  8    TABLESPACE indx02
  9    , PARTITION c_g_3 VALUES LESS THAN  (MAXVALUE)
 10    TABLESPACE indx03
 11    )  ;

Index created.
```

3. Create a local index on the partitioned CUSTS table.

   Index: Name CST_LC_FN. Index on the column FIRSTNAME.

   Partition: Partition names are irrelevant.

   Storage: Use tablespace INDX04.

```
SQL> CREATE INDEX cst_lc_fn
  2    ON custs ( firstname )
  3     TABLESPACE indx04
  4     LOCAL ;

Index created.
```

3b: Could you have specified anything else about the partition type or partition key values?

Answer: No. A local index follows the partitioning type and key values of the table. You can specify partition names and storage attributes.

4: Without referring to the USER_PART_INDEXES view, but possibly by examining other views, determine if this local index is prefixed or not. Afterwards, check your answer by selecting from USER_PART_INDEXES.ALIGNMENT.

Answer: Examining the partition key definition, to determine if it is the same as the index key (or the leading part thereof).

```
SQL> SELECT NAME, COLUMN_NAME, OBJECT_TYPE,
  2   COLUMN_POSITION "COL.POS."
  3    FROM USER_PART_KEY_COLUMNS
  4    WHERE NAME='CST_LC_FN' ;

NAME                           COLUMN_NAME     OBJECT_TYPE   COL.POS.
------------------------------ --------------- ----------- ----------
CST_LC_FN                      COUNTRY         INDEX                1

SQL> SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME, COLUMN_POSITION
  2    FROM USER_IND_COLUMNS
  3    WHERE INDEX_NAME='CST_LC_FN' ;

INDEX_NAME       TABLE_NAME   COLUMN_NAME     ColPos
---------------- ------------ --------------- ------
CST_LC_FN        CUSTS        FIRSTNAME            1
```

```
SQL> SELECT INDEX_NAME, ALIGNMENT from USER_PART_INDEXES
  2    WHERE INDEX_NAME='CST_LC_FN' ;

INDEX_NAME       ALIGNMENT
---------------- ------------
CST_LC_FN        NON_PREFIXED
```

5. Create a local partitioned index on SHIPPED.

   Index: Name: SHP_LC_PI. Normal index on column PROD_ID

   Storage: Specify nothing, use all defaults.

```
SQL> CREATE INDEX shp_lc_pi
  2   ON shipped ( prod_id )
  3    LOCAL ;

Index created.
```

6. Where should the local index partitions be stored: in the user default, the table default, or the current table partitions storage? After considering your answer, check it in the data dictionary.

```
SQL> SELECT INDEX_NAME, PARTITION_NAME, PARTITION_POSITION,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS
  4    WHERE INDEX_NAME='SHP_LC_PI' ;

INDEX_NAME       Part.Name          P.Pos HIGH_VALUE STATUS   TABLESPACE
---------------- ------------------ ----- ---------- -------- ----------
SHP_LC_PI        SHP_Q1_2002            1 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q2_2002            2 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q3_2002            3 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q4_2002            4 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q1_2003            5 TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2003            6 TIMESTAMP' USABLE   DATA03
```

Answer: The third option; Each local index partition has the same storage attributes as its corresponding table partition.

7. Create a global index on SHIPPED.

Index: Name SHP_GL_AM. Index on the column AMOUNT.

Partition: Range partition. Only one possible column can be the partition range key. Name the partitions: S_G_1 and S_G_2, with the partition key value for the first partition at 10. There is only one workable value for the second partition key value.

Storage: Use tablespaces INDX01 and INDX02.

```
SQL> CREATE INDEX shp_gl_am
  2    ON shipped ( amount )
  3    GLOBAL
  4    PARTITION BY RANGE ( amount )
  5   ( PARTITION s_g_1 VALUES LESS THAN ( 10 )
  6    TABLESPACE indx01
  7   , PARTITION s_g_2 VALUES LESS THAN ( MAXVALUE )
  8    TABLESPACE indx02
  9   ) ;

Index created.
```

8. Create a partitioned local bitmap index on SHIPPED.

   Index: Name: TST_LB_TI. Type: Bitmap. Index the columns TEST_ID.

   Storage: All defaults.

```
SQL> CREATE BITMAP INDEX tst_lb_ti
  2   ON test_result ( test_id )
  3   LOCAL ;

Index created.
```

9. A bitmapped global partitioned index is attempted on the hash-partitioned table TEST_RESULT. Will it succeed, and if not what is the failure reason? Try it.

   Index: Name TST_LB_BN. Type: Bitmap: Index on the column BATCH_NO.

   Partition: Range partition. Partitions to be named T_G_1 and T_G_2, with the partition key value for the first partition at 10.

   Storage: Use tablespaces INDX01 and INDX02.

```
SQL> CREATE BITMAP INDEX tst_lb_bn
  2   ON test_result ( batch_no )
  3    GLOBAL
  4    PARTITION BY RANGE ( batch_no )
  5   ( PARTITION t_g_1 VALUES LESS THAN ( 10 )
  6   TABLESPACE indx03
  7   , PARTITION t_g_2 VALUES LESS THAN ( MAXVALUE )
  8   TABLESPACE indx04
  9   ) ;
CREATE BITMAP INDEX tst_lb_bn
*
ERROR at line 1:
ORA-25113: GLOBAL may not be used with a bitmap index
```

   *Answer: Bitmap indexes can only be locally partitioned. You can create a B*tree global range partitioned index on a hash partitioned table.*

10. Use the data dictionary views to verify that your indexes are partitioned as expected. List partition key values and tablespace used as appropriate.

```
SQL> REM All Indexes
SQL> SELECT INDEX_NAME, INDEX_TYPE, UNIQUENESS, STATUS,
  2    TABLESPACE_NAME, PARTITIONED
  3    FROM USER_INDEXES ;

INDEX_NAME       INDEX_TYPE      Uq. STATUS   TABLESPACE PAR
---------------- --------------- --- -------- ---------- ---
CST_GL_LFN       NORMAL          NON N/A                 YES
CST_LC_FN        NORMAL          NON N/A                 YES
SHP_GL_AM        NORMAL          NON N/A                 YES
SHP_LC_PI        NORMAL          NON N/A                 YES
```

```
SHP_NP_CI        NORMAL          NON VALID    USERS      NO
SYS_IL0000010411 LOB             UNI N/A                 YES
C00003$$
TST_LB_TI        BITMAP          NON N/A                 YES

7 rows selected.
```

```
SQL> REM Partitioned Indexes
SQL> SELECT INDEX_NAME, PARTITIONING_TYPE,
  2    LOCALITY LOC, ALIGNMENT, PARTITION_COUNT,
  3    PARTITIONING_KEY_COUNT, DEF_TABLESPACE_NAME
  4    FROM USER_PART_INDEXES ;

INDEX_NAME       PARTITI LOC    ALIGNMENT     P.Cnt  P.K.# Def.TS Nam
---------------- ------- ------ ------------ ------ ------ ----------
CST_GL_LFN       RANGE   GLOBAL PREFIXED          3      1 USERS
CST_LC_FN        LIST    LOCAL  NON_PREFIXED      4      1 INDX04
SHP_GL_AM        RANGE   GLOBAL PREFIXED          2      1 USERS
SHP_LC_PI        RANGE   LOCAL  NON_PREFIXED      6      1
SYS_IL0000010411 HASH    LOCAL  NON_PREFIXED      8      1
C00003$$
TST_LB_TI        HASH    LOCAL  PREFIXED          8      1

6 rows selected.
```

```
SQL> REM Index Partitions
SQL> SELECT INDEX_NAME, PARTITION_NAME, PARTITION_POSITION,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS ;

INDEX_NAME       Part.Name          P.Pos HIGH_VALUE STATUS   TABLESPACE
---------------- ------------------ ----- ---------- -------- ----------
SYS_IL0000010411 SYS_IL_P1503           1            USABLE   DATA01
C00003$$
    :                                             :
SYS_IL00000104.. SYS_IL_P1510           8            USABLE   DATA02
C00003$$
CST_GL_LFN       C_G_1                  1 'H'        USABLE   INDX01
CST_GL_LFN       C_G_2                  2 'Q'        USABLE   INDX02
CST_GL_LFN       C_G_3                  3 MAXVALUE   USABLE   INDX03
CST_LC_FN        CUST_NA                1 'US', 'CA' USABLE   INDX04
CST_LC_FN        CUST_SA                2 'AR', 'BR' USABLE   INDX04
CST_LC_FN        CUST_EU                3 'DE', 'FR' USABLE   INDX04
CST_LC_FN        CUST_XX                4 'AU', 'IN' USABLE   INDX04
SHP_LC_PI        SHP_Q1_2002            1 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q2_2002            2 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q3_2002            3 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q4_2002            4 TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q1_2003            5 TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2003            6 TIMESTAMP' USABLE   DATA03
SHP_GL_AM        S_G_1                  1 10         USABLE   INDX01
SHP_GL_AM        S_G_2                  2 MAXVALUE   USABLE   INDX02
TST_LB_TI        H_1                    1            USABLE   DATA01
```

```
TST_LB_TI        H_2                    2           USABLE   DATA01
TST_LB_TI        H_3                    3           USABLE   DATA01
   :                                           :

31 rows selected.
```

```
SQL> REM Index Partition Keys
SQL> SELECT NAME INDEX_NAME, COLUMN_NAME,
  2    COLUMN_POSITION
  3    FROM USER_PART_KEY_COLUMNS
  4    WHERE TRIM(OBJECT_TYPE)!='TABLE'
  5    ;

INDEX_NAME       COLUMN_NAME     ColPos
---------------- --------------- ------
CST_GL_LFN       LASTNAME             1
CST_LC_FN        COUNTRY              1
SHP_GL_AM        AMOUNT               1
SHP_LC_PI        DATETIME             1
SYS_IL0000010411 TEST_ID              1
C00003$$
TST_LB_TI        TEST_ID              1

6 rows selected.
```

```
SQL> REM Index Key
SQL> SELECT INDEX_NAME, TABLE_NAME, COLUMN_NAME, COLUMN_POSITION
  2    FROM USER_IND_COLUMNS ;

INDEX_NAME       TABLE_NAME   COLUMN_NAME     ColPos
---------------- ------------ -------------- ------
CST_GL_LFN       CUSTS        LASTNAME            1
CST_GL_LFN       CUSTS        FIRSTNAME           2
CST_LC_FN        CUSTS        FIRSTNAME           1
SHP_NP_CI        SHIPPED      CUST_ID             1
SHP_LC_PI        SHIPPED      PROD_ID             1
SHP_GL_AM        SHIPPED      AMOUNT              1
TST_LB_TI        TEST_RESULT  TEST_ID             1

7 rows selected.
```

Output slightly reformatted

Practice 3-2: Specifying partitioned constraints

1. Attempt to add a unique key constraint, CST_LUQ_CI, to the CUSTS table on the CUST_ID column. The unique index created to support the constraint is to be local partitioned. Why does this fail?

```
SQL> ALTER TABLE custs ADD
  2    CONSTRAINT cst_luq_ci UNIQUE ( cust_id )
  3    USING INDEX LOCAL ;
ALTER TABLE custs ADD
*
ERROR at line 1:
ORA-14039: partitioning columns must form a subset of key columns of a
UNIQUE index
```

Answer: A local unique index must contain the partition columns (but not necessarily prefixed).

2. Extend the unique constraint definition so it can be locally partitioned.

```
SQL> ALTER TABLE custs ADD
  2    CONSTRAINT cst_luq_ci UNIQUE ( cust_id, country )
  3    USING INDEX LOCAL ;

Table altered.
```

Note that the index is not prefixed, it merely contains the partitioning columns.

3. Examine the partition names and storage location.

```
SQL> SELECT PARTITION_NAME, TABLESPACE_NAME
  2    FROM USER_IND_PARTITIONS
  3    WHERE INDEX_NAME='CST_LUQ_CI' ;

Part.Name          TABLESPACE
------------------ ----------
CUST_EU            DATA04
CUST_XX            DATA04
CUST_NA            DATA04
CUST_SA            DATA04
```

4. Create a global partitioned constraint on the hash-partitioned TEST_RESULT table.

   Constraint and supporting index: Name: TST_GUQ_BN. Column: BATCH_NO.

   Partitioning: Range partitioned on BATCH_NO, partition key values at 100, 200 and MAXVALUE

   Storage: Defaults

```
SQL> ALTER TABLE test_result ADD
  2    CONSTRAINT tst_guq_bn UNIQUE ( batch_no )
  3    USING INDEX
  4    TABLESPACE USERS GLOBAL
  5    PARTITION BY RANGE ( batch_no )
  6    ( PARTITION t_g_1 VALUES LESS THAN  (100)
  7    , PARTITION t_g_2 VALUES LESS THAN  (200)
  8    , PARTITION t_g_3 VALUES LESS THAN  (MAXVALUE)
  9    ) ;

Table altered.
```

Practice 4-1 Drop and Add table partition, with index maintenance

1. Another season has gone by, and it is time to do the Rolling Window Operation on the SHIPPED table. Drop the SHP_Q1_2002 partition, without any index maintenance.

```
SQL> ALTER TABLE shipped DROP PARTITION shp_q1_2002 ;

Table altered.
```

2. Examine the index status of all indexes on SHIPPED.

```
SQL> SELECT INDEX_NAME, INDEX_TYPE, UNIQUENESS, STATUS,
  2    TABLESPACE_NAME, PARTITIONED
  3    FROM USER_INDEXES
  4    WHERE TABLE_NAME='SHIPPED' ;

INDEX_NAME       INDEX_TYPE      Uq. STATUS   TABLESPACE PAR
---------------- --------------- --- -------- ---------- ---
SHP_GL_AM        NORMAL          NON N/A                 YES
SHP_LC_PI        NORMAL          NON N/A                 YES
SHP_NP_CI        NORMAL          NON UNUSABLE USERS      NO
```

```
SQL> SELECT INDEX_NAME, PARTITION_NAME,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS
  4    WHERE INDEX_NAME LIKE 'SHP%' ;

INDEX_NAME       Part.Name          HIGH_VALUE STATUS   TABLESPACE
---------------- ------------------ ---------- -------- ----------
SHP_LC_PI        SHP_Q2_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q3_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q4_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q1_2003        TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2003        TIMESTAMP' USABLE   DATA03
SHP_GL_AM        S_G_1              10         UNUSABLE INDX01
SHP_GL_AM        S_G_2              MAXVALUE   UNUSABLE INDX02
```

3. Attempt the following insert.

```
INSERT INTO shipped VALUES
   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 1234 ) ;
```

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 1234 ) ;
INSERT INTO shipped VALUES
*
ERROR at line 1:
ORA-01502: index 'DATAMGR.SHP_NP_CI' or partition of such index is in
unusable state
```

4. Fix the global index invalid status preventing the insert, and then attempt the insert again.

```
SQL> ALTER INDEX shp_np_ci REBUILD ;

Index altered.
```

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 1234 ) ;
INSERT INTO shipped VALUES
*
ERROR at line 1:
ORA-01502: index 'DATAMGR.SHP_GL_AM' or partition of such index is in
unusable state
```

5. Instead of fixing all partitions of the index in the last error message, only rebuild the S_G_2 partition.

```
SQL> ALTER INDEX shp_gl_am REBUILD ;
ALTER INDEX shp_gl_am REBUILD
            *
ERROR at line 1:
ORA-14086: a partitioned index may not be rebuilt as a whole
```

A global partitioned index must be rebuilt one partition at a time.

```
SQL> ALTER INDEX shp_gl_am REBUILD PARTITION s_g_2 ;

Index altered.
```

6. Attempt the above insert again. Attempt it also with the value 2.22 in the last column, AMOUNT. Commit the successful insert.

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 2.22 ) ;
INSERT INTO shipped VALUES
*
ERROR at line 1:
ORA-01502: index 'DATAMGR.SHP_GL_AM' or partition of such index is in
unusable state
```

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 1234 ) ;

1 row created.

SQL> COMMIT ;

Commit complete.
```

7. Check if the same partial index errors occur on queries. Query the table twice on AMOUNT having values 1234 and 2.22, respectively.

```
SQL> SELECT * FROM SHIPPED WHERE AMOUNT=1234 ;

   PROD_ID    CUST_ID  DATETIME           QUANTITY     AMOUNT
---------- ----------  -----------------  -----------  ------------
      2847       5190  05-MAY-02 00.00..            1          1234
```

```
SQL> SELECT * FROM SHIPPED WHERE AMOUNT=2.22 ;
SELECT * FROM SHIPPED WHERE AMOUNT=2.22
*
ERROR at line 1:
ORA-01502: index 'DATAMGR.SHP_GL_AM' or partition of such index is in
unusable state
```

Note that no tables or indexes have been analyzed.

8. Fix any remaining indexes so the insert with the value 2.22 also succeeds and commit it.

```
SQL> ALTER INDEX shp_gl_am REBUILD PARTITION s_g_1 ;

Index altered.
```

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2002-05-05 00:00:00.00', 1, 2.22 ) ;

1 row created.

SQL> COMMIT ;

Commit complete.
```

9. Having dropped and discarded the old data in step 1 above, you must make room for the new data. Add another partition to the SHIPPED table, continuing the pattern of partition attributes.

```
SQL> ALTER TABLE shipped ADD
  2  PARTITION shp_q3_2003 VALUES LESS THAN
  3   (TIMESTAMP '2003-10-01 00:00:00.00 +00:00')
  4   TABLESPACE data03 ;

Table altered.
```

10. Examine index status.

```
SQL> SELECT INDEX_NAME, INDEX_TYPE, UNIQUENESS, STATUS,
  2    TABLESPACE_NAME, PARTITIONED
  3    FROM USER_INDEXES
  4    WHERE TABLE_NAME='SHIPPED' ;

INDEX_NAME       INDEX_TYPE      Uq. STATUS   TABLESPACE PAR
---------------- --------------- --- -------- ---------- ---
SHP_GL_AM        NORMAL          NON N/A                 YES
SHP_LC_PI        NORMAL          NON N/A                 YES
SHP_NP_CI        NORMAL          NON VALID    USERS      NO
```

```
SQL> SELECT INDEX_NAME, PARTITION_NAME,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS
  4    WHERE INDEX_NAME LIKE 'SHP%' ;

INDEX_NAME       Part.Name          HIGH_VALUE STATUS   TABLESPACE
---------------- ------------------ ---------- -------- ----------
SHP_LC_PI        SHP_Q3_2003        TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q3_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q4_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q1_2003        TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2003        TIMESTAMP' USABLE   DATA03
SHP_GL_AM        S_G_1              10         USABLE   INDX01
SHP_GL_AM        S_G_2              MAXVALUE   USABLE   INDX02
```

11. Make the following inserts. Note the date or quarter of each insert.

```
INSERT INTO shipped VALUES
 ( 2847, 5190, TIMESTAMP '2002-02-02 00:00:00.00', 1, 1234 ) ;

INSERT INTO shipped VALUES
 ( 2847, 5190, TIMESTAMP '2003-09-09 00:00:00.00', 1, 1234 ) ;

INSERT INTO shipped VALUES
 ( 2847, 5190, TIMESTAMP '2003-11-11 00:00:00.00', 1, 1234 ) ;
```

Which inserts should fail? Which might have an undesirable effect? Commit inserts.

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2002-02-02 00:00:00.00', 1, 1234 ) ;

1 row created.
```

This belongs to the dropped partition. Adding a check constraint to the table would prevent it.

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2003-09-09 00:00:00.00', 1, 1234 ) ;

1 row created.
```

This makes use of the new partition, and is as expected.

```
SQL> INSERT INTO shipped VALUES
  2   ( 2847, 5190, TIMESTAMP '2003-11-11 00:00:00.00', 1, 1234 ) ;
INSERT INTO shipped VALUES
            *
ERROR at line 1:
ORA-14400: inserted partition key does not map to any partition
```

This is an expected failure, the date is too far in the future.

```
SQL> COMMIT ;

Commit complete.
```

Practice 4-2: Split and merge a partitioned table

1. Examine the table CUSTS. Because the volume of data is too skewed, you decide that the countries need to be rearranged by partition.

```
SQL> SELECT country, COUNT(country)
  2    FROM custs
  3    GROUP BY country ;

 CO COUNT(COUNTRY)
 -- --------------
 CA              2
 US          14172
 AR            253
 BR            759
 DE           8041
 DK            353
 ES           1986
 FR           3751
 IE           1958
 NL           7563
 UK           7475
 AU            767
 IN            676
 JP            593
 MY            570
 NZ            222
```

(Hash subpartitioning of list partitions is not supported in Oracle9*i*) You decide the following: Move the CA customers from the CUST_NA (North America) to CUST_SA (South America). This requires splitting and merging. Also, the partition that now contains only US customers is to be named CUST_USA, and the other American partition will be CUST_AM.

Merge the CUST_NA and CUST_SA into CUST_TMP in the DATA01 tablespace, as a temporary measure. You want to avoid rebuilding the global partitioned index.

```
SQL> ALTER TABLE custs  MERGE
  2    PARTITIONS cust_na, cust_sa
  3    INTO PARTITION cust_tmp TABLESPACE data01
  4    UPDATE GLOBAL INDEXES ;

Table altered.
```

2. Check index status. Hint: all relevant indexes start with CST.

```
SQL> SELECT INDEX_NAME, PARTITION_NAME,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS
  4    WHERE INDEX_NAME LIKE 'CST%' ;

INDEX_NAME   Part.Name    HIGH_VALUE             STATUS   TABLESPACE
------------ ------------ ---------------------- -------- ----------
CST_LUQ_CI   CUST_EU      'DE', 'FR', 'UK', 'DK' USABLE   DATA04
                          , 'ES', 'IE', 'NL'
CST_LUQ_CI   CUST_XX      'AU', 'IN', 'JP', 'MY' USABLE   DATA04
                          , 'NZ', NULL
CST_LUQ_CI   CUST_TMP     'US', 'CA', 'AR', 'BR' UNUSABLE DATA01
CST_LC_FN    CUST_TMP     'US', 'CA', 'AR', 'BR' UNUSABLE INDX04
CST_GL_LFN   C_G_1        'H'                    USABLE   INDX01
CST_GL_LFN   C_G_2        'Q'                    USABLE   INDX02
CST_GL_LFN   C_G_3        MAXVALUE               USABLE   INDX03
CST_LC_FN    CUST_EU      'DE', 'FR', 'UK', 'DK' USABLE   INDX04
                          , 'ES', 'IE', 'NL'
CST_LC_FN    CUST_XX      'AU', 'IN', 'JP', 'MY' USABLE   INDX04
                          , 'NZ', NULL
```

3. Split the CUST_TMP into the desired CUST_USA and CUST_AM, placing them into tablespaces DATA03 and DATA04, respectively. “Forget” to maintain the global indexes.

```
SQL> ALTER TABLE custs SPLIT
  2    PARTITION cust_tmp VALUES ( 'US' ) INTO
  3      ( PARTITION cust_usa TABLESPACE data03
  4      , PARTITION cust_am  TABLESPACE data04 )
  5     ;

Table altered.
```

4. Check index status. Note the partition key values, and the local index status.

```
SQL> SELECT INDEX_NAME, PARTITION_NAME,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS
  4    WHERE INDEX_NAME LIKE 'CST%' ;

INDEX_NAME   Part.Name    HIGH_VALUE             STATUS   TABLESPACE
------------ ------------ ---------------------- -------- ----------
CST_LUQ_CI   CUST_EU      'DE', 'FR', 'UK', 'DK' USABLE   DATA04
                          , 'ES', 'IE', 'NL'
CST_LUQ_CI   CUST_XX      'AU', 'IN', 'JP', 'MY' USABLE   DATA04
                          , 'NZ', NULL
CST_LUQ_CI   CUST_USA     'US'                   UNUSABLE DATA03
CST_LUQ_CI   CUST_AM      'CA', 'AR', 'BR'       UNUSABLE DATA04
CST_LC_FN    CUST_USA     'US'                   UNUSABLE INDX04
CST_LC_FN    CUST_AM      'CA', 'AR', 'BR'       UNUSABLE INDX04
CST_GL_LFN   C_G_1        'H'                    UNUSABLE INDX01
CST_GL_LFN   C_G_2        'Q'                    UNUSABLE INDX02
CST_GL_LFN   C_G_3        MAXVALUE               UNUSABLE INDX03
CST_LC_FN    CUST_EU      'DE', 'FR', 'UK', 'DK' USABLE   INDX04
                          , 'ES', 'IE', 'NL'
CST_LC_FN    CUST_XX      'AU', 'IN', 'JP', 'MY' USABLE   INDX04
                          , 'NZ', NULL
```

5. The table SHIPPED_T appears to be too crowded in the last range partition, so you increase the number of subpartitions.

```
SQL> ALTER TABLE shipped_t MODIFY
  2   PARTITION shp_q2_2003 ADD SUBPARTITION ;

Table altered.
```

This command is repeated for the number of subpartitions you want to add.

6. Because no storage specification was made, the subpartition ended up in the default tablespace, which is not the intention. Identify and move the subpartition to DATA04.

```
SQL> ALTER TABLE shipped_t MOVE
  2   SUBPARTITION sys_subp1234
  3   TABLESPACE data04 ;
   SUBPARTITION sys_subp1234
```

Practice 4-3: Exchange partition and table

1. Another season has passed, and it is time for the next rolling window operation of SHIPPED. However, an analyst wants to perform an in-depth analysis of the data in the SHP_Q2_2002 partition that you are about to discard, and asks that it be provided as a separate table.

   Create a suitable table, called OLD_SHIPPED in the USERS tablespace. Create an index on OLD_SHIPPED.PROD_ID.

```
SQL> CREATE TABLE old_shipped
  2    AS SELECT * FROM shipped
  3    WHERE ROWNUM < 1 ;

Table created.
```

```
SQL> CREATE INDEX shp_tab
  2    ON old_shipped ( prod_id ) ;

Index created.
```

2. Exchange SHP_Q2_2002 and OLD_SHIPPED, with the index.

```
SQL> ALTER TABLE shipped EXCHANGE
  2    PARTITION shp_q2_2002
  3    WITH TABLE old_shipped
  4    INCLUDING INDEXES ;

Table altered.
```

3. What is the status of the involved data now, specifically:

3a. Which tablespace is the old SHIPPED data, now in the OLD_SHIPPED table, located?

Answer: Because it has not been moved, it is in the same place as before,, that is, in the production tablespace DATAnn.

3b. Can the old data be queried through the SHIPPED table?

Answer: No

3c. If you now drop the SHP_Q2_2002 partition, will there be any unexpected side effects??

Answer: No. It should be as empty as the OLD_SHIPPED was before the exchange.

3d. How might you get the old data out of the production tablespaces (DATAnn)?

Answer1:`ALTER TABLE OLD_SHIPPED MOVE` *…*

Answer2:`ALTER TABLE MOVE PARTITION SHP_Q2_2002` *… before doing the exchange*

```
SQL> SELECT TABLE_NAME, PARTITION_NAME,
  2    PARTITION_NAME, TABLESPACE_NAME
  3    FROM USER_TAB_PARTITIONS
  4    WHERE TABLE_NAME LIKE '%SHIPPED' ;

TABLE_NAME   Part.Name          Part.Name          TABLESPACE
------------ ------------------ ------------------ ----------
SHIPPED      SHP_Q2_2002        SHP_Q2_2002        USERS
SHIPPED      SHP_Q3_2002        SHP_Q3_2002        DATA02
SHIPPED      SHP_Q4_2002        SHP_Q4_2002        DATA02
SHIPPED      SHP_Q1_2003        SHP_Q1_2003        DATA03
SHIPPED      SHP_Q2_2003        SHP_Q2_2003        DATA03
SHIPPED      SHP_Q3_2003        SHP_Q3_2003        DATA03
```

4. Check the status of the indexes on both OLD_SHIPPED and SHIPPED.

```
SQL> SELECT INDEX_NAME, INDEX_TYPE, UNIQUENESS, STATUS,
  2    TABLESPACE_NAME, PARTITIONED
  3    FROM USER_INDEXES
  4    WHERE TABLE_NAME like '%SHIPPED' ;

INDEX_NAME       INDEX_TYPE      Uq. STATUS   TABLESPACE PAR
---------------- --------------- --- -------- ---------- ---
SHP_GL_AM        NORMAL          NON N/A                 YES
SHP_LC_PI        NORMAL          NON N/A                 YES
SHP_NP_CI        NORMAL          NON UNUSABLE USERS      NO
SHP_TAB          NORMAL          NON VALID    DATA02     NO

SQL>
SQL> SELECT INDEX_NAME, PARTITION_NAME,
  2    HIGH_VALUE, STATUS, TABLESPACE_NAME
  3    FROM USER_IND_PARTITIONS
  4    WHERE INDEX_NAME LIKE 'SHP%' ;

INDEX_NAME       Part.Name          HIGH_VALUE STATUS   TABLESPACE
---------------- ------------------ ---------- -------- ----------
SHP_LC_PI        SHP_Q3_2003        TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2002        TIMESTAMP' USABLE   USERS
SHP_LC_PI        SHP_Q3_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q4_2002        TIMESTAMP' USABLE   DATA02
SHP_LC_PI        SHP_Q1_2003        TIMESTAMP' USABLE   DATA03
SHP_LC_PI        SHP_Q2_2003        TIMESTAMP' USABLE   DATA03
SHP_GL_AM        S_G_1              10         UNUSABLE INDX01
SHP_GL_AM        S_G_2              MAXVALUE   UNUSABLE INDX02
```

Practice 5-1 Export and Import of Partition

This exercise demonstrates the use of Export and Import with partitioned tables and should be performed as user `sh`. Export the 1998 Q1 partition. Name the export dump file `sales_q1_1998.dmp` and make sure it resides in your home directory. Perform a query that accesses data in this partition, then truncate the `sales_q1_1998` partition. Use Import to restore the data.

1. Connect as user `sh` and confirm the SALES table partition names.

```
SQL> Connect sh/sh
SQL> select partition_name from user_tab_partitions
  2    where table_name = 'SALES';

PARTITION_NAME
------------------------------
SALES_Q1_1998
SALES_Q2_1998
SALES_Q3_1998
SALES_Q4_1998
SALES_Q1_1999
SALES_Q2_1999
SALES_Q3_1999
SALES_Q4_1999
SALES_Q1_2000
SALES_Q2_2000
SALES_Q3_2000

PARTITION_NAME
------------------------------
SALES_Q4_2000

12 rows selected.
```

2. Perform the export. Make sure the dump file is written to your home directory.

```
$ exp sh/sh tables = sales:sales_q1_1998 file = $HOME/sales_q1_1998.dmp

Export: Release 9.0.1.0.0 - Production on Fri Jan 11 13:55:10 2002
Oracle9i Enterprise Release 9.0.1.0.0  With Partitioning option
Export done in US7ASCII character set and AL16UTF16 NCHAR character set
server uses WE8ISO8859P1 character set (possible charset conversion)

About to export specified tables via Conventional Path ...
. . exporting table                          SALES
. . exporting partition                  SALES_Q1_1998      71805 rows
exported
EXP-00091: Exporting questionable statistics.
...
Export terminated successfully with warnings.
```

3. Perform a query that accesses data in the SALES_Q1_1998 partition.

```
SQL> select prod_id, cust_id from sh.sales where
  2  time_id = '01-MAR-1998';

PROD_ID    CUST_ID
---------- ----------
     40690      42570
      1265     138090
     17035      35840
     12605      35640
      9015      26690
     11265      11850
      ...
   PROD_ID    CUST_ID
---------- ----------
      2555      27430
      3975      54230

750 rows selected.
```

4. Truncate the data in the partition SALES_Q1_1998.

```
SQL> alter table sales truncate partition sales_q1_1998;

Table altered.
```

5. Verify that the data is gone.

```
SQL> select prod_id, cust_id from sh.sales where
  2  time_id = '01-MAR-1998';

no rows selected
```

6. Import the data back into the empty partition.

```
$ imp sh/sh tables = sales:sales_q1_1998 ignore=y
file=$HOME/sales_q1_1998.dmp
...
Export file created by EXPORT:V09.00.01 via conventional path
import done in US7ASCII character set and AL16UTF16 NCHAR char set
import server uses WE8ISO8859P1 character set (possible charset
conversion)
. importing SH's objects into SH
. . importing partition    "SALES":"SALES_Q1_1998"      71805 rows
imported
Import terminated successfully without warnings.
```

7. Repeat the same query executed previously to verify that the data has been restored.

```
SQL> select prod_id, cust_id from sh.sales where
  2  time_id = '01-MAR-1998';

PROD_ID    CUST_ID
---------- ----------
     40690      42570
      1265     138090
     17035      35840
     12605      35640
      9015      26690
     11265      11850
      ...
   PROD_ID    CUST_ID
---------- ----------
      2555      27430
      3975      54230

750 rows selected.
```

Practice 5-2: Load a partition with SQL*Loader

This practice demonstrates how SQL*Loader works with partitioned tables. As user sh, truncate the SALES_Q1_1998 partition from the SALES table. The partition data will be loaded from the `sh_sales.dat file` located in `$ORACLE_HOME/demo/schema/sales_history` directory. Using the `sh_sales.ctl` control file as a model, create your own SQL*Loader control file in your home directory and reload the SALES_Q1_1998 partition.

1. Truncate the data in the SALES_Q1_1998 partition.

```
SQL> connect sh/sh
SQL> alter table sales truncate partition sales_q1_1998;Table altered.
```

2. Verify that the partition is empty.

```
SQL> select prod_id, cust_id from sh.sales where
  2  time_id = '01-MAR-1998';

no rows selected
```

3. Make sure you are in your home directory. Copy the `sh_sales`.ctl file to `sales.ctl` and make the necessary edits.

```
$ cd
$ cp $ORACLE_HOME/demo/schema/sales_history/sh_sales.ctl  sales.ctl
$ vi sales.ctl

LOAD DATA
APPEND
INTO TABLE sales partition (sales_q1_1998)
FIELDS TERMINATED BY "|"
( PROD_ID, CUST_ID, TIME_ID, CHANNEL_ID, PROMO_ID,
 QUANTITY_SOLD, AMOUNT_SOLD)
~
~
```

4. Use SQL*Loader to load the data into the partition SALES_Q1_1998 partition.

```
$ sqlldr sh/sh control = $HOME/sales.ctl log = $home/sales.log \
  data = $ORACLE_HOME/demo/schema/sales_history/sh_sales.dat \
  rows=10000

SQL*Loader: Release 9.0.1.0.0 - Production on Sat Jan 12 00:24:11 2002

(c) Copyright 2001 Oracle Corporation.  All rights reserved.

Commit point reached - logical record count 141
Commit point reached - logical record count 282
Commit point reached - logical record count 423
Commit point reached - logical record count 564
Commit point reached - logical record count 705
Commit point reached - logical record count 846
Commit point reached - logical record count 987
```

```
Commit point reached - logical record count 1128
Commit point reached - logical record count 1269
Commit point reached - logical record count 1410
...
Commit point reached - logical record count 71746
Commit point reached - logical record count 71887
```

5. Verify that the data has been successfully loaded.

```
SQL> select prod_id, cust_id from sh.sales where
  2  time_id = '01-MAR-1998';

PROD_ID    CUST_ID
---------- ----------
     40690      42570
      1265     138090
     17035      35840
     12605      35640
      9015      26690
     11265      11850
       ...
   PROD_ID    CUST_ID
---------- ----------
      2555      27430
      3975      54230

750 rows selected.
```

Practice 5-3 Partitions in Transportable Tablespaces

This exercise demonstrates self-containment of partitioned tables in transportable tablespaces. Perform all steps of this exercise as `sysdba`. Any transportable tablespace candidate must be self-contained. Perform a self-containment check of the tablespace SAMPLE. Then move the SALES partition SALES_Q1_1998 to the USERS tablespace. Perform another self-containment check and observe the differences.

1. Check for self-containment, using the `dbms_tts.transport_set_check` procedure.

```
SQL> connect / as sysdba
SQL> EXECUTE dbms_tts.transport_set_check ('SAMPLE');

PL/SQL procedure successfully completed.
```

2. View any violations by querying the `TRANSPORT_SET_VIOLATIONS` table.

```
SQL> SELECT * FROM TRANSPORT_SET_VIOLATIONS;

VIOLATIONS
--------------------------------------------------------------------------------
Snapshot SH.CAL_MONTH_SALES_MV in tablespace SAMPLE not allowed in transportable set
Snapshot SH.CAL_MONTH_SALES_MV in tablespace SAMPLE not allowed in transportable set
Snapshot SH.FWEEK_PSCAT_SALES_MV in tablespace SAMPLE not allowed in transportable set
Snapshot SH.FWEEK_PSCAT_SALES_MV in tablespace SAMPLE not allowed in transportable set

VIOLATIONS
--------------------------------------------------------------------------------
Master table SH.TIMES in tablespace SAMPLE not allowed in transportable set
Master table SH.PRODUCTS in tablespace SAMPLE not allowed in transportable set

6 rows selected.
```

3. Give user `sh` unlimited quota on the USERS tablespace and move the SALES_Q1_PARTITION:

```
SQL> alter user sh quota unlimited on users;
User altered.

SQL> alter table sh.sales move partition sales_q1_1998
tablespace users;

Table altered.
```

4. Rerun the self-containment check:

```
SQL> EXECUTE dbms_tts.transport_set_check('SAMPLE',TRUE);

PL/SQL procedure successfully completed.
```

5. Check again for violations.

```
SQL> SELECT * FROM TRANSPORT_SET_VIOLATIONS;

VIOLATIONS
--------------------------------------------------------------------------------
Partitioned Global index SH.SALES_CHANNEL_BIX in tablespace SAMPLE points to par
tition SALES_Q1_1998 of table SH.SALES in tablespace USERS outside of transporta
ble set

Partitioned Global index SH.SALES_CHANNEL_BIX in tablespace SAMPLE points to par
tition SALES_Q1_1998 of table SH.SALES in tablespace USERS outside of transporta
ble set

Partitioned Global index SH.SALES_CHANNEL_BIX in tablespace SAMPLE points to par
tition SALES_Q1_1998 of table SH.SALES in tablespace USERS outside of transporta
ble set

VIOLATIONS
--------------------------------------------------------------------------------
Partitioned Global index SH.SALES_CHANNEL_BIX in tablespace SAMPLE points to par
tition SALES_Q1_1998 of table SH.SALES in tablespace USERS outside of transporta
ble set

SAMPLE points to partition SALES_Q1_1998 of table SH.SALES in tablespace USERS ou
tside of transportable set

...
```

Practice 6-1 Rolling Window Operation

This exercise emphasizes the mechanics of performing rolling-window operations. Our attention will be focused on the fact table SALES in the SH schema. It has now become necessary to drop the oldest partition, SALES_q1_1998, and add a brand new SALES_q1_2001 partition. Perform the necessary steps to accomplish this task. Don't forget about index maintenance.

1. Connect as SH and query the partitions currently comprising the SALES table.

```
SQL> connect sh/sh
SQL> select partition_name from user_tab_partitions
   2 where table_name = 'SALES';

PARTITION_NAME
------------------------------
SALES_Q1_1998  (partition to drop)
SALES_Q2_1998
SALES_Q3_1998
SALES_Q4_1998
SALES_Q1_1999
SALES_Q2_1999
SALES_Q3_1999
SALES_Q4_1999
SALES_Q1_2000
SALES_Q2_2000
SALES_Q3_2000

PARTITION_NAME
------------------------------
SALES_Q4_2000

12 rows selected.
```

2. Drop the partition SALES_Q1_1998.

```
SQL> alter table sales
  2  drop partition sales_q1_1998;

Table altered.
```

3. Add another partition SALES_Q1_2001 above the partition SALES_Q4_2000. Since that partition is bounded by MAXVALUE, you must split SALES_Q4_2000.

```
SQL> alter table sales
  2  split partition SALES_Q4_2000
  3  at (to_date('01-JAN-2001', 'DD-MON-YYYY')) into
  4  ( partition SALES_Q4_2000, partition SALES_Q1_2001 );

Table altered.
```

4. Check to see that the new SALES_Q1_2001 partition has been properly created.

```
SQL> select partition_name from user_tab_partitions
   2 where table_name = 'SALES';

PARTITION_NAME
------------------------------
SALES_Q2_1998
SALES_Q3_1998
SALES_Q4_1998
SALES_Q1_1999
SALES_Q2_1999
SALES_Q3_1999
SALES_Q4_1999
SALES_Q1_2000
SALES_Q2_2000
SALES_Q3_2000
SALES_Q4_2000
SALES_Q1_2001  (new partition)
```

5. The indexes for the fact table SALES must reflect the fact that you have dropped one partition and added another. Query the USER_PART_INDEXES view to determine the associated indexes for the table.

```
SQL> select index_name from user_part_indexes
  2  where table_name = 'SALES';

INDEX_NAME
------------------------------
SALES_CHANNEL_BIX
SALES_CUST_BIX
SALES_PROD_BIX
SALES_PROMO_BIX
SALES_TIME_BIX
```

6. Identify the index partitions to be rebuilt. Select the index partition_name from the USER_IND_PARTITIONS view.

```
SQL> select partition_name from user_ind_partitions
   2 where index_name = 'SALES_CHANNEL_BIX';

PARTITION_NAME
------------------------------
SALES_Q2_1998
...
SALES_Q4_2000
SALES_Q1_2001

11 rows selected.

SQL> select partition_name from user_ind_partitions where index_name =
'SALES_CUST_BIX';
```

```
PARTITION_NAME
------------------------------
SALES_Q2_1998
...
SALES_Q4_2000
SALES_Q1_2001


SQL> select partition_name from user_ind_partitions
   2 where index_name = 'SALES_PROD_BIX';

PARTITION_NAME
------------------------------
SALES_Q2_1998
...
SALES_Q4_2000
SALES_Q1_2001


SQL> select partition_name from user_ind_partitions
   2 where index_name = 'SALES_PROMO_BIX';

PARTITION_NAME
------------------------------
SALES_Q2_1998
...
SALES_Q4_2000
SALES_Q1_2001


SQL> select partition_name from user_ind_partitions
   2 where index_name = 'SALES_TIME_BIX';

PARTITION_NAME
------------------------------
SALES_Q2_1998
...
SALES_Q4_2000
SALES_Q1_2001

12 rows selected.
```

7. Rebuild the affected indexes.

```
SQL> ALTER INDEX SALES_CHANNEL_BIX REBUILD PARTITION SALES_Q4_2000;
Index altered.

SQL> ALTER INDEX SALES_CHANNEL_BIX REBUILD PARTITION SALES_Q1_2001;
Index altered.

SQL> ALTER INDEX SALES_CUST_BIX REBUILD PARTITION SALES_Q4_2000;
Index altered.

SQL> ALTER INDEX SALES_CUST_BIX REBUILD PARTITION SALES_Q1_2001;
```

```
Index altered.

SQL> ALTER INDEX SALES_PROD_BIX REBUILD PARTITION SALES_Q4_2000;
Index altered.

SQL> ALTER INDEX SALES_PROD_BIX REBUILD PARTITION SALES_Q1_2001;
Index altered.

SQL> ALTER INDEX SALES_PROMO_BIX REBUILD PARTITION SALES_Q4_2000;
Index altered.

SQL> ALTER INDEX SALES_PROMO_BIX REBUILD PARTITION SALES_Q1_2001;
Index altered.

SQL> ALTER INDEX SALES_TIME_BIX REBUILD PARTITION SALES_Q4_2000;
Index altered.

SQL> ALTER INDEX SALES_TIME_BIX REBUILD PARTITION SALES_Q1_2001;
Index altered.
```

Practice 6-2 Partitioned View to Partitioned Table Conversion

In this exercise, you will create a partition view and then complete the steps required to convert it to a partitioned table. As user `sh`, create three standard tables as select * from sales, partitions SALES_Q1_1999 through SALES_Q1_1999 inclusive. Create a partitioned view called SALES_PART_VIEW from the three newly created tables. Run the `$HOME/STUDENT/LABS/lab_06_02_view_to_table.sql` script to create an empty partitioned table called SALES_PART_TABLE. Exchange each partition with its corresponding table.

1. Create tables.

```
SQL> connect sh/sh
SQL> create table q1_1999_sales as
  2  select * from sales partition (sales_q1_1999);
Table created.

SQL> create table q2_1999_sales as
  2  select * from sales partition (sales_q2_1999);
Table created.

SQL> create table q3_1999_sales as
  2  select * from sales partition (sales_q3_1999);
Table created.
```

2. Create the partitioned view. Connect as SYSDBA and grant `create view` to the user sh to accomplish this.

```
SQL> connect / as sysdba
SQL> grant create view to sh;
SQL> connect sh/sh

SQL> create view sales_part_view as
  2  select * from sh.q1_1999_sales
  3  union all
  4  select * from sh.q2_1999_sales
  5  union all
  6  select * from sh.q3_1999_sales;

View created.
```

3. Prepare for the migration by creating the partitioned table SALES_PART_TABLE. You can create it by running the script `$HOME/STUDENT/LABS/lab_06_02_view_to_table.sql`. Please inspect this script before you execute it. It will be empty in anticipation of the migrated data, so notice that a segment of two blocks is specified as an initial storage value to act as a placeholder.

```
SQL> !cat $HOME/STUDENT/LABS/lab_06_02_view_to_table.sql
CREATE TABLE sales_part_table
    ( prod_id        NUMBER(6)
        CONSTRAINT   sale_product_nn     NOT NULL
    , cust_id        NUMBER
        CONSTRAINT   sales_customer_nn    NOT NULL
    , time_id        DATE
        CONSTRAINT   sale_time_nn        NOT NULL
    , channel_id     CHAR(1)
        CONSTRAINT   sale_channel_nn     NOT NULL
    , promo_id       NUMBER(6)
        CONSTRAINT   sales_promo_nn       NOT NULL
    , quantity_sold  NUMBER(3)
        CONSTRAINT   sale_quantity_nn    NOT NULL
    , amount_sold         NUMBER(10,2)
        CONSTRAINT   sale_amount_nn      NOT NULL
    )  TABLESPACE sample STORAGE (INITIAL 2)
        PARTITION BY RANGE (time_id)
  (PARTITION SALES_Q1_1999 VALUES LESS THAN
      (TO_DATE('01-APR-1999','DD-MON-YYYY')),
  PARTITION SALES_Q2_1999 VALUES LESS THAN
      (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
  PARTITION SALES_Q3_1999 VALUES LESS THAN (MAXVALUE))
;
SQL> connect sh/sh
SQL>@$HOME/STUDENT/LABS/lab_06_02_view_to_table.sql
Table created.
```

4. Use the EXCHANGE PARTITION statement to migrate the tables to the corresponding partitions.

```
SQL> alter table SALES_PART_TABLE
  2  exchange partition sales_q1_1999 with table
  3  sh.q1_1999_sales with validation;
Table altered.

SQL> alter table SALES_PART_TABLE
  2  exchange partition sales_q2_1999 with table
  3  sh.q2_1999_sales with validation;
Table altered.

SQL> alter table SALES_PART_TABLE
  2  exchange partition sales_q3_1999 with table
  3  sh.q3_1999_sales with validation;
Table altered.
```

5. In the real world, you would then drop the original partitioned view and use the old view name to rename the new partitioned table so that the change would be transparent to the users.

```
SQL> drop view sales_part_view
View dropped.

SQL> rename sales_part_table to sales_part_view
Table renamed.
```

Practice 6-3 A Very Mixed Table

In this exercise, you will execute the `lab_06_03_create_mix.sql` script located in the `$HOME/STUDENT/LABS` directory to create a table that will demonstrate partitioned table support of various data types, data organization, constraints, and so on. The table is called MIX and creates the following columns and datatypes:

NU – NUMBER
CH – CHAR
VC – VARCHAR
CL – CLOB
BL – BLOB
TS - TIMESTAMP

NU and VC are primary keys while CH and VC are unique. The table is range partitioned on the VC column. The MIX table uses tablespaces DATA01 through DATA04 and INDEX01 through INDEX04 for storage, both primary and overflow. Two local indexes are created, one on TS and another on VC and TS.

Spend a few moments and inspect the `lab_06_03_create_mix.sql` script. Pay special attention to the column datatypes, partitioning statements, storage parameters, constraints, and index creation.

```
REM create Partition Table
REM

DROP TABLE mix ;

CREATE TABLE mix
/* --- Column defenitions, thus relational --- */
  ( nu NUMBER(6)
       /*CONSTRAINT mix_pk PRIMARY KEY*/
  , ch CHAR(10)
       /*CONSTRAINT mix_uq UNIQUE*/
  , vc VARCHAR2(20)
       CONSTRAINT mix_ck CHECK ( LENGTH(vc)>5 )
  , cl CLOB
  , bl BLOB
  , CONSTRAINT mix_pk UNIQUE ( nu, vc )
  , CONSTRAINT mix_uq UNIQUE ( ch, vc )
  , ts TIMESTAMP(2)
  )
/*
 --- plain or index organized
*/
ORGANIZATION HEAP
/*
  --- Nested or vararray section ---
*/
/*   none  */
/*
```

```
  --- Lob attributes ---
*/
LOB ( cl ) STORE AS mix_cl
  ( TABLESPACE data04
    DISABLE STORAGE IN ROW
  )
LOB ( bl ) STORE AS mix_bl
  ( /* TABLESPACE data04 */
    DISABLE STORAGE IN ROW
  )
/*
  --- Physical attributes of table ---
*/
TABLESPACE indx04
PCTFREE 5
-- PCTTHRESHOLD 20  >>IOT
-- OVERFLOW TABLESPACE index01  >> IOT
/*
  --- Partition clauses ---
*/
PARTITION BY RANGE ( vc ) /* alternativly SUBPARTITION; HASH; LIST */
  ( PARTITION mix_p1 VALUES LESS THAN ( 'A' )
        TABLESPACE data02
        PCTFREE 10
        -- OVERFLOW TABLESPACE indx02
        LOB ( cl ) STORE AS mix_cl_p1
          ( DISABLE STORAGE IN ROW )
        LOB ( bl ) STORE AS mix_bl_p1
          ( ENABLE STORAGE IN ROW
            TABLESPACE data02
          )
  , PARTITION mix_p2 VALUES LESS THAN ( ']' /* chr(asc('Z')+1) */ )
        TABLESPACE data03
        PCTFREE 10
        -- OVERFLOW TABLESPACE indx02
        LOB ( cl ) STORE AS mix_cl_p2
          ( DISABLE STORAGE IN ROW )
        LOB ( bl ) STORE AS mix_bl_p2
          ( ENABLE STORAGE IN ROW
            /* TABLESPACE data02 */
          )
  , PARTITION mix_p3 VALUES LESS THAN ( '~' )
        TABLESPACE data03
        PCTFREE 10
        -- OVERFLOW TABLESPACE indx02
        LOB ( cl ) STORE AS mix_cl_p3
          ( DISABLE STORAGE IN ROW )
        LOB ( bl ) STORE AS mix_bl_p3
          ( ENABLE STORAGE IN ROW
            /* TABLESPACE data02  */
          )
  , PARTITION mix_p4 VALUES LESS THAN ( MAXVALUE )
        TABLESPACE data03
        PCTFREE 10
```

```
        -- OVERFLOW TABLESPACE indx02
        LOB ( cl ) STORE AS mix_cl_p4
          ( DISABLE STORAGE IN ROW )
        LOB ( bl ) STORE AS mix_bl_p4
          ( ENABLE STORAGE IN ROW
            TABLESPACE data02
          )
  )
ENABLE ROW MOVEMENT
/*--- Constraints, indexes used with ---*/
ENABLE CONSTRAINT mix_pk USING INDEX
  GLOBAL PARTITION BY RANGE ( nu )
    ( PARTITION mix_pk_p1 VALUES LESS THAN ( 0 )
        TABLESPACE indx02
    , Partition mix_pk_p2 VALUES LESS THAN ( MAXVALUE )
        TABLESPACE indx03
    )
ENABLE CONSTRAINT mix_uq USING INDEX
  LOCAL
;
/*Lets insert some values*/
INSERT INTO mix VALUES
  ( 1, 'Hello', 'This is a test', empty_clob(), empty_blob(),
LOCALTIMESTAMP ) ;
DECLARE
  clob_loc CLOB;
  txt_buff VARCHAR2(1000) ;
BEGIN
 SELECT cl INTO clob_loc FROM mix WHERE nu=1 /*not FOR UPDATE*/ ;
 txt_buff := RPAD('OneThousand characters', 1000, 'bla ') ;
 FOR i IN 1..10 LOOP
  DBMS_LOB.WRITEAPPEND (clob_loc, 1000, txt_buff );
  END LOOP;
END;
/
DECLARE
  blob_loc BLOB;
  raw_buff RAW(1000) ;
BEGIN
 SELECT bl INTO blob_loc FROM mix WHERE nu=1 FOR UPDATE ;
 raw_buff := HEXTORAW(RPAD('1000', 2000, '1000') ) ;
 FOR i IN 1..10 LOOP
  DBMS_LOB.WRITEAPPEND (blob_loc, 1000, raw_buff );
  END LOOP;
END;
/

INSERT INTO mix VALUES
  ( 2, 'Hi there', 'continuation of test', NULL, NULL, LOCALTIMESTAMP ) ;
UPDATE mix SET
   cl=(SELECT cl FROM mix WHERE nu=1 ),
   bl=(SELECT bl FROM mix WHERE nu=1 )
 WHERE nu=2;
```

```
INSERT INTO mix
  SELECT -2, 'Hi again', '... test', cl,bl, LOCALTIMESTAMP
    FROM mix WHERE nu = 2 ;

CREATE INDEX mix_ts1 ON mix ( ts ) LOCAL ;
CREATE INDEX mix_ts2 ON mix ( vc, ts ) LOCAL ;
```

1. Execute the script `$HOME/STUDENT/LABS/lab_06_03_create_mix.sql`.

```
SQL> connect system/manager
SQL> @$HOME/STUDENT/LABS/lab_06_03_create_mix.sql
Table dropped.
Table created.
1 row created.
PL/SQL procedure successfully completed.
PL/SQL procedure successfully completed.
1 row created.
1 row updated.
1 row created.
Index created.
Index created.
```

2. Check table and partition creation.

```
SQL> select partition_name,tablespace_name from dba_tab_partitions
  2  where table_name = 'MIX';

PARTITION_NAME                 TABLESPACE_NAME
------------------------------ ------------------------------
MIX_P1                         DATA02
MIX_P2                         DATA03
MIX_P3                         DATA03
MIX_P4                         DATA03
```

3. Look at the partitioned columns.

```
SQL> select column_name, object_type, column_position
  2  from dba_part_key_columns where name = 'MIX';

COLUMN_NAM OBJECT_TYPE COLUMN_POSITION
---------- ----------- ---------------
VC         TABLE                     1
```

4. Look at the indexes associated with the MIX table.

```
SQL> select index_name, index_type, partitioned
  2  from dba_indexes where table_name = 'MIX';

INDEX_NAME                     INDEX_TYPE                  PAR
------------------------------ --------------------------- ---
MIX_PK                         NORMAL                      YES
SYS_IL0000005177C00005$$       LOB                         YES
SYS_IL0000005177C00004$$       LOB                         YES
MIX_UQ                         NORMAL                      YES
MIX_TS1                        NORMAL                      YES
MIX_TS2                        NORMAL                      YES
```