



## CHAPTER 5

# *Partitioning administration*

### **QUICK Reference**

#### *If you want to...*

#### **A. Do partitioned table administration**

- Create range partitioned table... see 5.1
- Convert to a partitioned table... see 5.2
- Manage partitioned table... see 5.5
- Monitor partitioned table... see 5.6
- Partition pruning... see 5.7
- Hash partitioned table... see 5.11
- Composite partitioned table... see 5.12
- Merge partitions... see 5.13
- Row movement between partitions... see 5.14

#### **B. Do partitioned index administration**

- Local partition index... see 5.8
- Global partition index... see 5.9
- Monitor partition index... see 5.10

#### **C. Do partition view administration**

- Partition view... see 5.3
- Convert partition view to partitioned table... see 5.4

## OVERVIEW

Partitioning is one of the most useful features that has become available with Oracle8. It gives you the functionality to break a large table or index into smaller units for better manageability, availability, and increased performance. Unlike a partition view that requires rewriting application code, partitioning does not. It is application-independent, and almost all of the Oracle tools and commands are partition-aware, including the cost-based optimizer. Partitioning also provides for high availability for the tables. If one of the partitions is not available, you can still query the remaining online partitions. Partitioning is especially useful for data warehousing applications that support very large tables and indexes.

In Oracle8 there was originally only one method by which you could partition a table, and that was range partitioning. Oracle8i has now introduced two more methods: hash partitioning and composite partitioning. In range partitioning, each partition is assigned an upper and a lower limit key value that controls where a row will be inserted. In hash partitioning the row is inserted using a hash function. Rows are mapped to the partitions based on the hash value of the partitioning key. Composite partitioning uses a combination of both range partitioning and hash partitioning. It first partitions the data using the range method, and within each partition it applies the hash method.

The advantages of partitioning include reduced downtime for maintenance and disk failure, support of very large tables and indexes, higher performance on queries, partition elimination, and it is application-independent.

## QUESTIONS

### 5.1 *How do I create a range-partitioned table?*

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition table
<b>Operating system:</b> All	

#### **PROBLEM**

I have a very large table that I am having difficulty backing up. I know that in Oracle8 there is a partitioning feature that allows you to back up a partition. How do I create a partitioned table?

#### **SOLUTION**

Partitioning is a new feature available in Oracle8 which allows large tables to be broken down into smaller units for better manageability, availability, and performance. Each unit or partition can have different physical attributes, i.e., can be stored on different tablespaces, but must have the same logical structure, i.e., must have the same column name, data types, constraints, and even triggers. Each of the partitions is stored in a separate segment. By allowing each partition

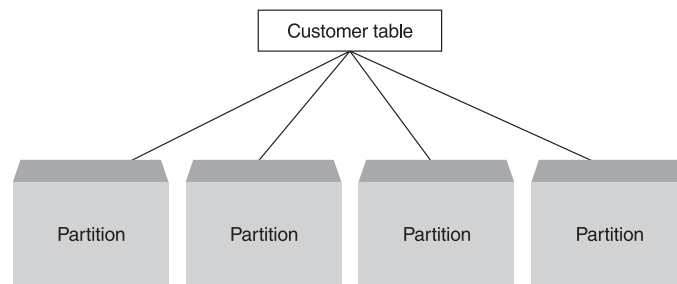
to be stored on a different tablespace, Oracle provides for greater availability. If one of the partitions is not available, you can still use the other partitions.

Partitioning a table provides many benefits, including:

- Ease of backup and recovery
- Improved query performance
- Ability to load and unload a partition
- Better manageability of the table
- Transparency to the application

In range partitioning, each partition is assigned an upper and a lower limit key value that controls where a row will be inserted. These are specified by the `VALUES LESS THAN` clause associated with each partition. An important point to note when creating a partitioned table is that if a row does not fall into any of the partitions, then that row will be rejected with Oracle error ORA-14400 which reads “inserted partition key is beyond highest legal partition key.” To avoid such a problem, Oracle provides the `MAXVALUE` option to handle rows that do not fall into any of the partitions.

In addition to regular tables, you can also partition indexes for nonclustered tables. Index partitioning does require that each partition have the same logical structure (columns), but they can have different physical attributes (different tablespaces). The indexed key determines the partition upon which that index entry will be stored. The indexed key can consist of one or more of the table’s columns. The layout of a basic partitioned table is diagrammed in figure 5.1.



**Figure 5.1**  
**Partitioned table**

### **STEPS**

**Step 1—Creating a partitioned table.** To create a partitioned table, you can use the `CREATE TABLE` command (listing 5.1) with the `PARTITION` option.

```
CREATE TABLE <table_name>
  ( {column-name data-type [,] } )
  PARTITION BY RANGE ( { partition-column [,] } )
    (PARTITION partition-name VALUES LESS THAN ( value-list )
     [TABLESPACE Tablespace-name] .... )
```

**Listing 5.1** Create table SQL syntax with partitioning option

In the example that follows, we will be creating a CUSTOMER table with the customer number as the primary key (listing 5.2). We will be partitioning the table based on the STATE-CODE column with four partitions. Each partition will be stored on a separate tablespace. For this example we have already created the four tablespaces STATE1, STATE2, STATE3 and STATE4.

```

SQL> create table customer
2      ( cust_number  NUMBER PRIMARY KEY,
3        name         CHAR(40),
4        address_1   CHAR(40),
5        address_2   CHAR(40),
6        state_code  CHAR(2),
7        zip_code    NUMBER )
8  PARTITION BY RANGE(state_code)
9      (PARTITION S1 VALUES LESS THAN ('IA')
10         TABLESPACE STATE1
11         STORAGE (INITIAL 1M NEXT 1M),
12        PARTITION S2 VALUES LESS THAN ('MO')
13         TABLESPACE STATE2
14         STORAGE (INITIAL 1M NEXT 1M),
15        PARTITION S3 VALUES LESS THAN ('PA')
16         TABLESPACE STATE3
17         STORAGE (INITIAL 1M NEXT 1M),
18        PARTITION S4 VALUES LESS THAN (MAXVALUE)
19         TABLESPACE STATE4)
20         STORAGE (INITIAL 1M NEXT 1M);

```

Table created.  
SQL>

❶ Range for partition S1

❷ Range for partition S2

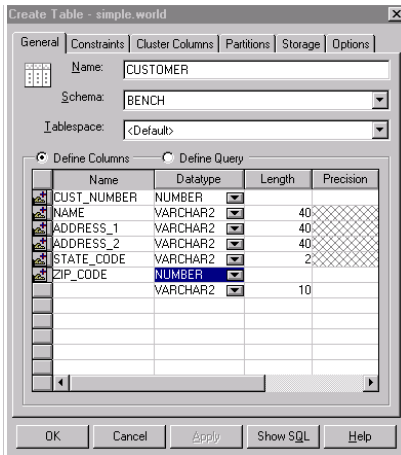
❸ Range for partition S3

❹ Range for partition S4

**Listing 5.2** Sample SQL statement to create a partitioned table

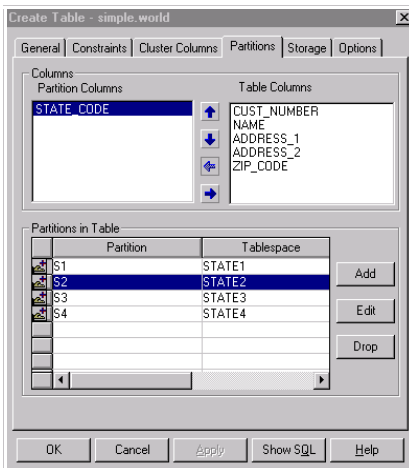
- ❶ For partition S1, we will specify all values between 'A' and 'IA'. This partition will be placed in tablespace STATE1.
- ❷ Anything beyond IA and less than MO will be stored in partition S2 under the tablespace STATE2.
- ❸ State code less than PA but greater than MO will be stored in partition S3.
- ❹ And finally partition S4 will store anything that is beyond state code PA which is represented by MAXVALUE.

To create a partitioned table using the Enterprise Manager, you would have to choose the Create Table Manually option in the New Table Creation menu. You cannot use the Table Wizard to create a partitioned table. Once you specify the Create Table Manually option, you are prompted with the Create Table dialog window as shown in figure 5.2.



**Figure 5.2**  
Create table dialog

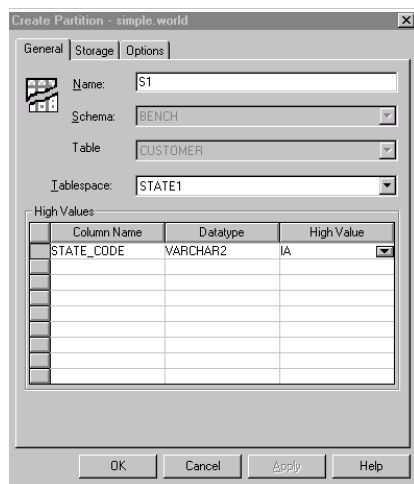
After you have defined the table schema, you will then need to click on the Partitions tab in the Create Table window to bring up the partition properties (Figure 5.3). In the partition properties dialog, you can choose the partition columns from the list of the table columns. You can have one or more of the table columns defined in the partition columns.



**Figure 5.3**  
The partition properties dialog

After defining your partitioned columns, you can add partitions to your table by clicking on the Add button in the partitions properties dialog. When you click on the Add button, you get the Create Partition window (figure 5.4). Here you need to enter the partition name and the tablespace name to associate with that partition. For each partition column you can enter the High Value for the partition. In our example, we had the high value set to be “IA” for partition S1. You can also define the extent size for each of the cluster partitions by using the storage properties (click on the Storage tab to

invoke the storage properties). Once you have defined your partitions, click on OK and the Create Partition window will disappear, taking you back to the partition properties window.



**Figure 5.4**  
The Create Partition window

**Step 2—Inserting data into the partitioned table.** Inserting data into a partitioned table requires no special consideration. Basically a partitioned table is transparent to the application. Let us now insert some rows into the partitioned table (listing 5.3).

```
SQL> insert into customer values (1001,'JOHN',
2      '1260 New Memory Ave', 'APT 901T1', 'CA',95691);
1 row created.
SQL> insert into customer values (1002,'NOEL',
2      '880 Old Storage Rd', ' ', 'TX',89691);
1 row created.
SQL>
```

**Listing 5.3** Inserting data into a partitioned table

---

**NOTE** If you try to insert data into a partitioned table that does not have the MAX-VALUE specified for one of the partitions, and the value you are trying to insert is greater than specified by your CREATE TABLE definition, then the insert will fail, and Oracle will generate an ORA-14400 error message.

---

Oracle does not allow you to update the partition key column if it will cause the row to migrate to another partition. You will receive the ORA-14402 error message “updating partition key column would cause a partition change.” The workaround for this is to delete the row from the partition and insert it into another, with the updated value. If you are using Oracle8i then look at question 5.14 for information on how to enable row movement between partitions.

**Step 3—Querying a partitioned table.** Listing 5.4 shows how to query a partitioned table. According to the CREATE TABLE partition definition, we stated that any customer whose state code is less than IA should go to partition S1, so customer number 1001 would be positioned in that partition. Noel, Customer 1002, would appear in partition S4 because he lives in Texas.

The first query retrieves both of the rows from the CUSTOMER table because it accesses rows from all of the partitions. The second query retrieves just one row from partition S1, because there is only one customer who lives in California. Queries numbered 3 and 4 do not retrieve any rows because there are no customers in state code greater than “IA” and less than “PA.” Refer to listing 5.2 to see how the customer table is partitioned. The last query (query 5), retrieves one row because there is a customer who lives in the state code “TX” (Texas).

**Query 1. Query all of the customer rows.**

```
SQL> select cust_number,name from customer;

CUST_NUMBER NAME
-----
          1001 JOHN
          1002 NOEL
```

**Query 2. Query only partition S1 from the customer table.**

```
SQL> select cust_number,name from customer partition (s1);

CUST_NUMBER NAME
-----
          1001 JOHN
```

**Query 3. Query only partition S2 from the customer table.**

```
SQL> select cust_number,name from customer partition (s2);
no rows selected
```

**Query 4. Query only partition S3 from the customer table.**

```
SQL> select cust_number,name from customer partition (s3);
no rows selected
```

**Query 5. Query only partition S4 from the customer table.**

```
SQL> select cust_number,name from customer partition (s4);

CUST_NUMBER NAME
-----
          1002 NOEL

SQL>
```

**Listing 5.4 Querying the partitioned table**

**Step 4—Testing for availability.** One of the greatest benefits of using partitioned tables is availability. In partitioning, if any one or more partitions become unavailable due to disk failure, file corruption, or the like, the other partitions can still be available for use.

Now let us simulate a tablespace failure by taking it offline as shown in listing 5.5. Once a tablespace is offline and we try to query that table, the query will fail with an Oracle error message of “ORA-00376: file cannot be read at this time.” If we queried a partition whose tablespace is offline it will also fail. The Oracle error would be the same ORA-00376. However, if we issue a query against a partition that is still available, we succeed in retrieving the information.

**Query 1. Take tablespace offline.**

```
SQL> Alter tablespace STATE4 offline;
Tablespace altered.
```

**Query 2. Query all of the customer rows.**

```
SQL> select cust_number,name from customer;
ERROR:
ORA-00376: file 7 cannot be read at this time
ORA-01110: data file 7: '/raid2/oracle/STATE2.DBF'
no rows selected
SQL>
```

**Query 3. Query customer table for the offline partition.**

```
SQL> select * from customer partition (S4);
select * from customer partition (S4)
      *
ERROR at line 1:
ORA-00376: file 16 cannot be read at this time
ORA-01110: data file 16: '/raid2/oracle/STATE2.DBF'
SQL>
```

**Query 4. Query customer table for available partition.**

```
SQL>select cust_number,name from customer partition (s1);

CUST_NUMBER NAME
-----
          1001 JOHN
SQL>
```

**Listing 5.5 Testing customer table for availability**

**CONCLUSION**

Partitioning is one of the best new features available in Oracle8. It takes away the complexity and problems associated with managing very large tables. It allows large tables to be broken down into smaller units for better manageability, availability, and performance. Oracle only supports table and index partitioning, it does not support partitioning for clustered tables and snapshots. Oracle8 only offers the range partitioning method, while Oracle8*i* offers hash partitioning and composite partitioning as well.

## 5.2 *How can I convert an existing table to a range partitioned table?*

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition table
<b>Operating system:</b> All	

### **PROBLEM**

I have just migrated to Oracle8 and would like to implement the new partitioning feature. How do I convert my existing tables to a range-partitioned table? What are the recommendations for selecting the tables to be partitioned?

### **SOLUTION**

Implementing the Oracle8 partitioning feature is not really simple. It requires some planning with the DBAs and application developers who have to decide which tables to partition and, most importantly, which partition key to use. Let us briefly take a look at these issues before we go about converting an existing table to a range partitioned table.

#### ***Which tables to partition?***

This is one of the questions that needs to be addressed before actually converting an existing table to a partitioned table. This feature has been introduced primarily to handle very large tables. With the advancement in technology and with DSS applications increasing, along with the ever falling prices of hard drives, large corporations have started building terabyte databases. These corporations have extremely large tables in the range of hundreds of gigabytes. With Oracle7, backing up such a large table is quite a challenge, even if we do not consider availability and performance. What partitioning permits is the breaking up of these very large tables into smaller units for better manageability, availability, and performance.

To take advantage of table partitioning, a table should be at least half a gigabyte or larger. Anything less than that can be implemented, but you will not see the real benefits of partitioning. After all, what good is it to partition, if a table can be cached entirely in memory? That then, could be your starting point in deciding which tables should be partitioned. Once you have picked your tables, you can then proceed to the next issue of “what partition key to use”?

#### ***Which columns should make up the partition key?***

This is an important question to address before actually implementing a conversion process or creating a new partitioned table. The ideal columns that should be considered for the partition key are the primary key, foreign key, or the DATE column. These should be columns that do not change often, since Oracle8 does not allow you to update a partition key, which will cause it to migrate to another partition. However, if you are using Oracle8i, it is supported (see question 5.14).

### **How should I store the table partitions?**

If possible you should store each of the partitions on a separate tablespace. This provides not only higher performance but also higher availability. If one of the datafiles in a tablespace becomes corrupted or the disk fails, chances are that you might lose only one partition as opposed to the entire table.

### **Converting an existing table to partitioned table**

For an existing table the ideal way to convert from a nonpartitioned to a partitioned table is by using the Export/Import utility. The Export utility allows you to export a nonpartitioned table and then import it back into a newly created partitioned table. The Export/Import utility is “partition aware” and automatically puts the row in the appropriate partitions. In Oracle8, you can export and import either by a table or by a partition. However, you cannot do a partition-level incremental export.

Although in Oracle8 there is a feature that allows you to exchange data segments between a table and a partition, this is only applicable to partition views. So if you have a partitioned view instead of a regular table you can use the EXCHANGE option which is fast and does not require you to export and import the table. Please refer to question 5.4 for more details on how to convert a partition view to a partitioned table using the EXCHANGE option.

## **STEPS**

**Step 1—Existing table.** Let us walk through the process of converting an existing table into a partitioned table. For our example we will use the table called CUSTOMER that has 6 columns and stores the customer information for a given company (listing 5.6).

```
create table customer
( cust_number  NUMBER PRIMARY KEY,
  name         CHAR(40),
  address_1   CHAR(40),
  address_2   CHAR(40),
  state_code  CHAR(2),
  zip_code    NUMBER )
```

### **Listing 5.6 Customer table structure**

Let us assume that there are a million rows in the CUSTOMER table. It is indexed on the CUST\_NUMBER that uniquely identifies the row. Based on this layout we can now proceed in converting this million-row table into a partitioned table.

**Step 2—Exporting data.** The first step would be to export the table using the Export utility. In this example we will export the CUSTOMER table (listing 5.7).

```
$ exp system/manager tables=(customer)

Export: Release 8.0.3.0.0 - Production on Mon May 4 20:25:21 1998
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Connected to: Oracle8 Enterprise Edition Release 8.0.3.0.0 - Production
```

```

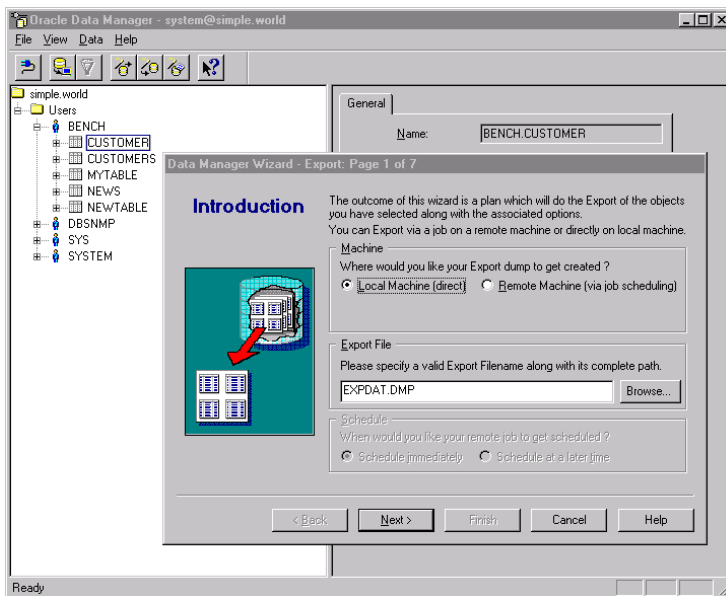
With the Partitioning and Objects options
PL/SQL Release 8.0.3.0.0 - Production
Export done in US7ASCII character set and US7ASCII NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table          CUSTOMER          1000000 rows exported
Export terminated successfully without warnings.
$

```

**Listing 5.7 Using the export utility**

You can also export the tables using the Oracle Data Manager in the Enterprise Manager software. The Oracle Data Manager allows you to export, import and load data into and out of the database. It provides a nice GUI interface and has Data Manager Wizard that walks you through the process of exporting/importing or loading data. The Oracle Data Manager is shown in figure 5.5.



**Figure 5.5 Oracle Data Manager**

Once we have exported the table, we would then have to drop the CUSTOMER table and recreate it as a partitioned table (listing 5.8). For the CUSTOMER table we have decided to use the STATE\_CODE as the partitioned key. We will have four partitions, each of which will be stored in a separate tablespace as shown in listing 5.8.

```

SQL> drop table customer;
Table dropped.
SQL> create table customer
2         ( cust_number  NUMBER PRIMARY KEY,
3           name          CHAR ( 40 ) ,
4           address_1    CHAR ( 40 ) ,

```

```

5          address_2      CHAR(40),
6          state_code     CHAR(2),
7          zip_code       NUMBER )
8  PARTITION BY RANGE(state_code)
9      (PARTITION S1 VALUES LESS THAN ('IA')
10         TABLESPACE STATE1
11         STORAGE (INITIAL 1M NEXT 1M),
12         PARTITION S2 VALUES LESS THAN ('MO')
13         TABLESPACE STATE2
14         STORAGE (INITIAL 1M NEXT 1M),
15         PARTITION S3 VALUES LESS THAN ('PA')
16         TABLESPACE STATE3
17         STORAGE (INITIAL 1M NEXT 1M),
18         PARTITION S4 VALUES LESS THAN (MAXVALUE)
19         TABLESPACE STATE4)
20         STORAGE (INITIAL 1M NEXT 1M);

```

Table created.

SQL>

### Listing 5.8 Creating a partitioned table

**Step 3—Importing the table.** Once we have recreated the CUSTOMER table, we can import it back into the partitioned table using the Import utility (listing 5.9). You must specify the IGNORE=Y option with the import command, because the table already exists in the database and we want to just load the data.

```

$ imp system/manager TABLES=(customer) IGNORE=Y
Import: Release 8.0.3.0.0 - Production on Mon May 4 20:38:43 1998
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Connected to: Oracle8 Enterprise Edition Release 8.0.3.0.0 - Production
With the Partitioning and Objects options
PL/SQL Release 8.0.3.0.0 - Production
Export file created by EXPORT:V08.00.03 via conventional path
. importing SYSTEM's objects into SYSTEM
. . importing table          "CUSTOMER"          1000000 rows imported
Import terminated successfully without warnings.
$

```

### Listing 5.9 Using the Import utility

The import should automatically load the rows into the correct partitions based on the partition key since the Import is partition-aware. Now you can query the CUSTOMER table and see if the rows are in the appropriate partitions.

## CONCLUSION

There is no easy way to convert an existing table into a partitioned table. The Export/Import utilities are partition-aware and can export and import partitions. If you were using a partitioned view in Oracle7.3, then Oracle8 allows you to convert that view into a partitioned table very quickly by exchanging the data segments, as discussed in question 5.4.

## 5.3 *How can I create a partition view?*

<b>Applies to:</b> Oracle7.3 or Higher, Oracle8	<b>CD Key:</b> Partition view
<b>Operating system:</b> All	

### **PROBLEM**

Currently I am using Oracle7.3 with a DSS application that has two very large tables. I am having difficulty backing up the tables. I know that in Oracle8 there is the new partitioning feature available to manage large tables. Is there something similar that I can implement in Oracle7.3?

### **SOLUTION**

In Oracle7.3 there is a feature called Partition View, which closely resembles the Partitioned Table. With this technique, you can build separate tables with similar logical structure and then define a view that creates a UNION of these tables. This newly defined view is what is called a Partition View. The partition view can help improve query performance by eliminating partitions. In Oracle7.3, the query optimizer is partitioned-view aware and therefore can dramatically speed up the retrieval of the data. However, partition view also comes with many drawbacks including:

#### **DDL limitation**

There can be no global index defined for a partition view. This is one of the biggest drawbacks. Basically the primary key cannot maintain uniqueness across the various partitions in the partition view.

#### **Manageability issue**

When you implement Partition View, you are on your own in managing the partitions for operations such as adding new partitions, merging, splitting, and so forth.

#### **Performance issue**

Even though the Optimizer is partition-aware, in Oracle7.3, it still has problems with some SQL queries and therefore does not take into account all of the base table indexes.

#### **Limited partition transparency**

Statements such as INSERT require a table name instead of view. Therefore the application needs to be coded according to the base tables.

Because of the preceding limitations, Oracle developed the partitioned table feature in version 8. If you are using Oracle8, it is recommended that you switch to a partitioned table instead of using partitioned view. (See question 5.4 for details on moving a partitioned view and converting it to a partitioned table.)

### **STEPS**

Following is an example of how to create a partitioned view in Oracle7.3 (see listing 5.10). Here we have two tables, one called ORDERS\_OPEN and the other

called `ORDERS_CLOSED`, each having similar logical structures (columns, data types) but different physical attributes, such as tablespace storage, and so forth. The `ORDERS_OPEN` table stores the orders that are currently open, while the `ORDERS_CLOSED` table contains records that relate to closed orders.

```

SVRMGR> CREATE TABLE orders_open
2>     ( order_number integer,
3>       customer_number integer,
4>       order_date date,
5>       order_value number,
6>       order_status number );
Statement processed.
SVRMGR> CREATE TABLE orders_closed
2>     ( order_number integer,
3>       customer_number integer,
4>       order_date date,
5>       order_value number,
6>       order_status number );
Statement processed.
SVRMGR> alter table orders_open
2>     add constraint c_orders_open
3>       check (order_status=1);
Statement processed.
SVRMGR> alter table orders_closed
2>     add constraint c_orders_closed
3>       check (order_status=0);
Statement processed.
SVRMGR> create view orders as
2> select * from
3>     orders_open union all
4> select * from
5>     orders_closed;
Statement processed.
SVRMGR>

```

**1 Create table orders\_open**

**2 Create table orders\_closed**

**3 Add constraint to orders\_open**

**4 Add constraint to orders\_closed**

**5 Create a view called orders**

**Listing 5.10 Creating partitioned view**

- 1** We first create a table called `ORDERS_OPEN` which has five columns.
- 2** We then create another table called `ORDERS_CLOSED`.
- 3** We add constraint to the `ORDERS_OPEN` table to insert only those orders with the `order_status` column set to 1.
- 4** We also add constraint to the `ORDERS_CLOSED` table by specifying to insert only those orders with the `ORDERS_STATUS` column set to 0.
- 5** And finally we create a view called `orders` that creates a `UNION` of both of the tables, namely the `ORDERS_OPEN` and `ORDERS_CLOSED` tables.

Also you have to set the `INIT.ORA` parameter called `PARTITION_VIEW_ENABLED` which has either a `TRUE` or a `FALSE` value, the default being `FALSE`.

When the parameter is set to TRUE, the optimizer skips unnecessary table accesses in the partition view. You may also enable this parameter using the ALTER SESSION command.

### CONCLUSION

Although the partitioned view feature does offer limited manageability, availability, and performance, it is not as robust as the partitioned table feature. The major shortcoming with a partitioned view is that you have to manage the partitions yourself, that is, the applications have to be rewritten to handle partitioning. Unless you are totally committed to Oracle7.3, to get the most out of partitioning you should migrate to Oracle8 or Oracle8i.

## 5.4 How can I convert a partition view to a partitioned table?

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition view
<b>Operating system:</b> All	

### PROBLEM

I have just migrated to Oracle8 and have a partition view from an existing database. I would like to convert this partition view to a partitioned table under Oracle8. How can I accomplish this quickly?

### SOLUTION

To convert a nonpartitioned table to a partitioned table, Oracle8 provides you with an ALTER TABLE statement with the EXCHANGE option. This option allows you to convert a nonpartitioned table to a partition by exchanging their data segments. This only works with tables and not with views, with the exception of partition views (tables joined using UNION ALL). Although any nonpartitioned table will work, the EXCHANGE option is only useful for partitioned views because it exchanges data segments between a partition and a table. So if you had a nonpartitioned table, then all of the data would go into one single partition. That defeats the partitioning purpose. Ideally you would have multiple tables (partition view), each table having specific data pertaining to a key; then the EXCHANGE option would work fine, because each of the tables would then move to the appropriate partition.

The EXCHANGE option basically allows you to convert a nonpartitioned table to a partitioned table by exchanging the data segments. Following is the syntax for the ALTER TABLE with the EXCHANGE option (listing 5.11):

```
ALTER TABLE <table_name>
    EXCHANGE PARTITION <partition-name>
        WITH TABLE <nonpartitioned table-name>
            [ INCLUDING/EXCLUDING INDEXES ]
            [ WITH/WITHOUT VALIDATION ]
```

**Listing 5.11** ALTER TABLE command with EXCHANGE option

The ALTER TABLE with the EXCHANGE PARTITION command is quite fast since it only exchanges data segments and does not copy the data. Along with the table conversion, the index can also be exchanged using the INCLUDING INDEXES option.

## STEPS

**Step 1—Table setup.** Before we can convert an existing table, there are a number of steps we must go through, as in the following example. For this example, we have two customer tables. The CUSTOMER\_1000 stores customer numbers that are less than 1000 while the CUSTOMER\_2000 table stores numbers that are greater than 1000. Each of the tables has a similar logical structure but different physical attributes. Also, we apply some constraints to the tables so that only valid customer numbers are stored in the respective tables (listing 5.12).

```
SQL> create table customer_1000
2      ( cust_number number PRIMARY KEY,
3        name          varchar2(40),
4        address_1     varchar2(40),
5        address_2     varchar2(40),
6        city          varchar2(40),
7        state         varchar2(2),
8        zip           number,
9        balance       number(7,2))
        tablespace customer_tblsp
        storage (initial 2M next 2M);
```

Table created.

```
SQL> create table customer_2000
2      ( cust_number number PRIMARY KEY,
3        name          varchar2(40),
4        address_1     varchar2(40),
5        address_2     varchar2(40),
6        city          varchar2(40),
7        state         varchar2(2),
8        zip           number,
9        balance       number(7,2));
        tablespace customer_tblsp
        storage (initial 2M next 2M);
```

Table created.

```
SQL> alter table customer_1000
2      add constraint cust_1000 check (cust_number<=1000);
```

Table altered

```
SQL> alter table customer_2000
2      add constraint cust_2000 check (cust_number>1000);
```

Table altered.

```
SQL> create view customer_part as
2      select * from customer_1000 union all
3      select * from customer_2000;
```

View created.

```
SQL> insert into customer_1000
```

**1 Create table customer\_1000**

**1 Create table customer\_2000**

**2 Add constraint**

**2 Add constraint**

**3 Create view**

**4 Insert rows**

```

2         values (1,'Noel Y','1234 Memory Lane',
3               'Apt JK1','Dallas','TX',34567,12050.08);
1 row created.
SQL> insert into customer_2000
2         values (1001,'Veronica V','734 Post Street',
3               ' ','Los World','CA',19507,45905.82);
1 row created.
SQL>

```

4 Insert rows

**Listing 5.12 Creating a partitioned view table**

- 1 First we create a table called CUSTOMER\_1000 which contains 8 columns and has CUST\_NUMBER as the primary key. We also create another table called CUSTOMER\_2000 with a similar structure.
- 2 We then add constraints to both of the tables. We only want customer numbers less than or equal to 1000 to be inserted into the CUSTOMER\_1000 table and anything beyond 1000 to go into the CUSTOMER\_2000 table.
- 3 We then create a view called CUSTOMER\_PART which joins the two tables.
- 4 And finally we insert some rows into both of the tables.

**Step 2—Creating a new partitioned table.** Now we are ready to create a new partitioned table called CUSTOMER (listing 5.13). It contains four partitions named S1 through S4 and is partitioned using the CUSTOMER\_NUMBER.

```

SQL> create table customer
2         ( cust_number number PRIMARY KEY,
3           name          varchar2(40),
4           address_1     varchar2(40),
5           address_2     varchar2(40),
6           city          varchar2(40),
7           state         varchar2(2),
8           zip           number,
9           balance       number(7,2))
10 PARTITION BY RANGE (cust_number)
11     (PARTITION CUST1 VALUES LESS THAN (1001)
12       TABLESPACE CUST_TBLSP1
13       STORAGE (INITIAL 1M NEXT 1M),
14     PARTITION CUST2 VALUES LESS THAN (MAXVALUE)
15       TABLESPACE CUST_TBLSP2
16       STORAGE (INITIAL 1M NEXT 1M) );
Table created.
SQL>

```

**Listing 5.13 Creating a partitioned table**

**Step 3. Doing the EXCHANGE of the data segments.** Once the tables are created, we can run the ALTER TABLE with the PARTITION option to exchange the data segments. But first, let us look at what blocks are being used by the CUSTOMER\_TBLSP tablespace. As we can see from listing 5.14, there are two data segments from file 9, each consisting of 1,025 blocks of segment type TABLE.

```
SQL> select header_file, header_block, bytes, blocks, segment_type
2     from dba_segments where
3         tablespace_name = 'CUSTOMER_TBSLSP';
```

HEADER_FILE	HEADER_BLOCK	BYTES	BLOCKS	SEGMENT_TYPE
9	2	2099200	1025	TABLE
9	1032	2099200	1025	TABLE
9	1027	10240	5	INDEX
9	2057	10240	5	INDEX

```
SQL>
```

**Listing 5.14 Data segments mapped to tablespace CUSTOMER\_TBSLSP**

Now let us look at the data segments from the newly created partitioned table. In the output shown in listing 5.15, we can see that we have two data segments from two different files, namely 10 and 11, each consisting of 515 blocks and with the segment type as TABLE PARTITION.

```
SQL> select header_file, header_block, bytes, blocks, segment_type
2     from dba_segments where
3         partition_name like 'CUST%';
```

HEADER_FILE	HEADER_BLOCK	BYTES	BLOCKS	SEGMENT_TYPE
11	2	1054720	515	TABLE PARTITION
10	2	1054720	515	TABLE PARTITION

```
SQL>
```

**Listing 5.15 Data segments mapped to partition CUSTOMER**

Now let us EXCHANGE the data segments between the table and the partition. To exchange them you would have to use the ALTER TABLE command with the EXCHANGE PARTITION option as shown in listing 5.16.

***DML 1. Exchanging table customer\_1000 with partition CUST1***

```
SQL> alter table customer EXCHANGE PARTITION CUST1
with TABLE CUSTOMER_1000;
Table altered.
SQL>
```

***DML 2. Exchanging table customer\_2000 with partition CUST2***

```
SQL> alter table customer EXCHANGE PARTITION CUST2
with TABLE CUSTOMER_2000;
Table altered.
SQL>
```

**Listing 5.16 Exchanging data segments between a table and a partition**

If we now query the DBA\_SEGMENTS table, we find that the data segments have been exchanged. The table report shows that it has two segments from file 9 with 1,025 blocks and has segment table TABLE PARTITION instead of files 10 and 11 with 515 blocks (listing 5.17).

**Query 1. Retrieving information on the data segments mapped to partition CUSTOMER**

```
SQL> select header_file, header_block, bytes, blocks, segment_type
2      from dba_segments where
3          partition_name like 'CUST%';
```

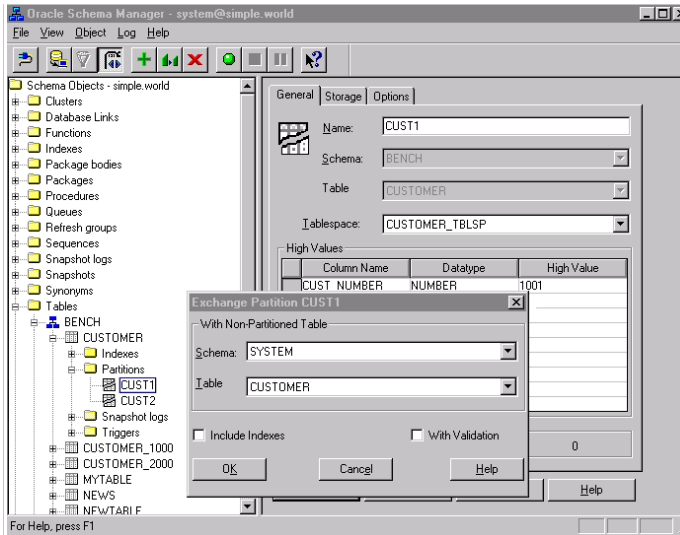
HEADER_FILE	HEADER_BLOCK	BYTES	BLOCKS	SEGMENT_TYPE
9	1032	2099200	1025	TABLE PARTITION
9	2	2099200	1025	TABLE PARTITION

SQL>

**Listing 5.17 Data segments mapped to partition CUSTOMER**

The EXCHANGE option works quite fast since only the data segments are exchanged not the data itself. Once you have made the exchange, you should back up the database.

You can also do the EXCHANGE using the Oracle Schema Manager available in OEM. In the Schema Manager, click on the Tables drop-down list, followed by the user-name with which the table is associated. If you right-click on the partition name you are then prompted with a floating menu. Choose the EXCHANGE option which will bring up the Exchange Partition dialog box. You can then choose the nonpartitioned table with which you would like to exchange the data segments. Figure 5.6 shows you the Schema Manager.



**Figure 5.6**  
**The Oracle**  
**Schema Manager**

**CONCLUSION**

The EXCHANGE option is quite powerful and allows you to make a nonpartitioned table into a partitioned table on the fly. However it does come with limitations. The EXCHANGE option only exchanges the data segments with one

partition and one table, that is, if you do not have a partitioned view then all of the rows end up in a single partition.

## 5.5 *How can I manage a range-partitioned table?*

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition table
<b>Operating system:</b> All	

### **PROBLEM**

I already have a partitioned table in Oracle8, but I am not sure how to manage the partitions. What commands are available to manage the partitions?

### **SOLUTION**

In Oracle8 there are some new options available in the ALTER TABLE statement that allow you to perform maintenance on the partitions. With these options you can ADD, MOVE, DROP, TRUNCATE, and SPLIT partitions.

Let us go through some of the options for managing the partitions.

#### **Add partitions**

You can add partitions by using the ADD PARTITION option (listing 5.18). However, this option only adds a new partition beyond the highest partition value. Thus if you have set a MAXVALUE for one of the partitions, you cannot use the ADD PARTITION option, unless you drop that partition.

```
ALTER TABLE <table_name>
ADD PARTITION <partition-name>
VALUES LESS THAN (Value_list)
```

**Listing 5.18 ALTER TABLE command with ADD PARTITION option**

---

**EXAMPLE** The following example (listing 5.19) shows that we have a PHONES table that is divided into four partitions, each on a separate tablespace. For the last partition we have used the MAXVALUE option that allows us to store rows that do not fall into any other partition.

---

```
SQL> create table phones
2      ( phone_number NUMBER,
3        person        CHAR(30),
4        phone_code    CHAR(1),
5        zip_code      NUMBER,
6        state_code    CHAR(2))
7 PARTITION BY RANGE(state_code)
8      (PARTITION S1
9        VALUES LESS THAN ('IA')
10       TABLESPACE STATE1,
11     PARTITION S2
12       VALUES LESS THAN ('MO')
13       TABLESPACE STATE2,
14     PARTITION S3
```

```

15             VALUES LESS THAN ('PA')
16             TABLESPACE STATE3,
17     PARTITION S4
18             VALUES LESS THAN (MAXVALUE)
19             TABLESPACE STATE4);

```

Table created.

SQL>

#### Listing 5.19 Creating a partitioned table called PHONES

Having created the PHONES table, we now try to ADD a partition that is higher than the PARTITION S3 value “PA” but less than the MAXVALUE. We get an ORA-14074 error, where the partition value should be higher than the last partition value. In this case we will not be able to accomplish the ADD because there is no higher value than MAXVALUE (listing 5.20).

```

SQL> alter table phones
2     ADD PARTITION S5 VALUES LESS THAN ('QA')
3     TABLESPACE STATE5;
      ADD PARTITION S5 VALUES LESS THAN ('QA')
      *

```

ERROR at line 2:

```

ORA-14074: partition bound must collate higher than that of the last
partition

```

#### Listing 5.20 Adding a partition

You can also use the Oracle Schema Manager to add a new partition for an existing partitioned table. Figure 5.7 shows the Create Partition dialog which adds a new partition named CUST3 to the CUSTOMER table. If you click on the Show SQL button you can also view the SQL statement generated by the Schema Manager.

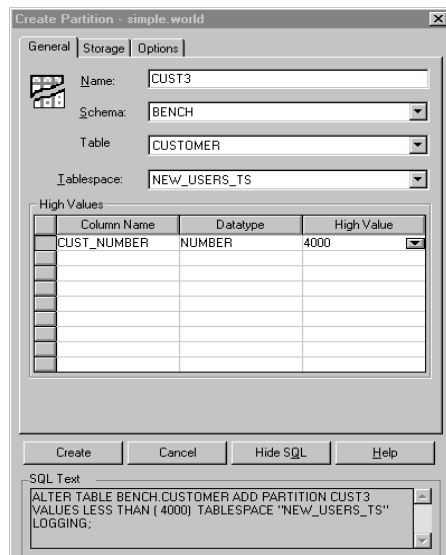


Figure 5.7  
Create Partition dialog

### ***Move partitions***

To move partitions from one tablespace to another, you can use the MOVE PARTITION option (listing 5.21). If you are having performance problems with a partition, you can move it to another segment without having to move the entire table.

```
ALTER TABLE <table_name>
MOVE PARTITION <partition-name>
    WITH TABLE <non-partitioned table-name>
    [ INCLUDING/EXCLUDING INDEXES ]
    [ WITH/WITHOUT VALIDATION]
```

#### **Listing 5.21 Move partition syntax**

Listing 5.22 shows an example of moving an EMP1 partition to a new tablespace—USERTS.

```
ALTER TABLE employee
    MOVE PARTITION EMP1
    TABLESPACE USERTS;
```

#### **Listing 5.22 Move partition example**

You can also do the move using the Oracle Schema Manager. Simply change the tablespace name in the Partition properties window.

### ***Drop partitions***

You can likewise drop a partition by exercising the DROP PARTITION option in the ALTER TABLE command (listing 5.23). You cannot drop all of the partitions in a table—at least one should remain. When you issue the DROP PARTITION command, it not only drops the partition but removes all of the rows. So take care when issuing the command.

```
ALTER TABLE <table_name>
DROP PARTITION <partition-name>
```

#### **Listing 5.23 Drop partition syntax**

The following example drops the partition S1:

```
ALTER TABLE employee
    DROP PARTITION S1;
```

#### **Listing 5.24 Drop partition example**

To drop the partition from the Schema Manager, simply right click on the selected partition and a floating menu appears. From the menu choose the Remove option and you can then drop the partition.

### ***Truncate partitions***

If you want to remove all of the rows from a partition, you can use the TRUNCATE PARTITION option in the ALTER TABLE command (listing 5.25). The option only removes the rows, and the partition remains intact.

```
ALTER TABLE <table_name>
TRUNCATE PARTITION <partition-name>
    DROP/REUSE
```

#### **Listing 5.25 Truncate partition syntax**

The following example truncates the S2 partition from the employee table.

```
ALTER TABLE employee
    TRUNCATE PARTITION S2;
```

**Listing 5.26 Truncate partition example**

To truncate a partition using the Oracle Schema Manager, right click on the partition name and choose the Truncate option.

**Split partitions**

When a partition becomes too large and the backup is taking longer, you can split the partition into two by using the SPLIT PARTITION option in the ALTER TABLE command. If you try to split the partition into more than two, you get the ORA-14046 error: “a partition may be split into exactly two new partitions.”

```
ALTER TABLE <table_name>
    SPLIT PARTITION <partition-name> AT (values_list)
    INTO (partition...)
```

**Listing 5.27 Split partition syntax**

The following example shows how to split a partition. After running this example, you would have two partitions named S41 and S42, and S4 would no longer exist.

```
ALTER TABLE employee
    SPLIT PARTITION S4 AT ('ST')
    INTO (PARTITION S41, PARTITION S42).
```

**Listing 5.28 Split partition example**

To use the Schema Manager to split partitions, right-click on the partition name, followed by choosing the Split option which will display the Split Partition window (figure 5.8). In this window you can choose the value at which to split the partitions.

**Renaming partitions**

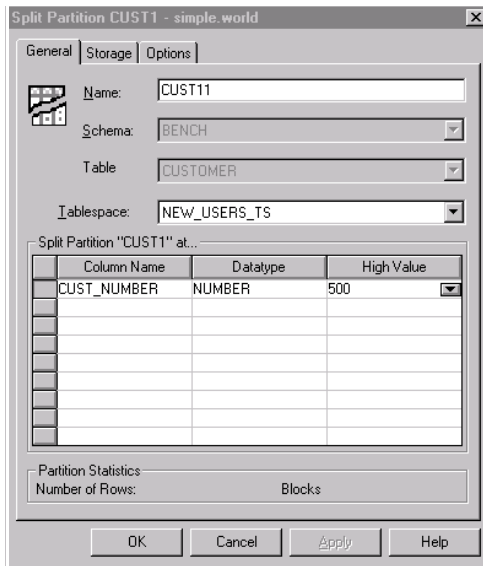
You can rename a partition using the ALTER TABLE command with the RENAME PARTITION option. I personally have not found much use for it, since the partition name has to be unique for a table or an object but not for the user or for the database, i.e., you can have the same partition name for different tables.

```
ALTER TABLE <table_name>
    RENAME PARTITION <partition_name> TO <new_partitionname>
```

**Listing 5.29 Syntax for renaming a partition**

**CONCLUSION**

Oracle8 provides you with many options to perform maintenance on the partitions. You can ADD, MOVE, DROP, SPLIT and TRUNCATE partitions very easily.



**Figure 5.8**  
Split Partition window

## 5.6 *How can I monitor the partitioned tables in Oracle8?*

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition table
<b>Operating system:</b> All	

### **PROBLEM**

I would like to find out what tables have been partitioned for a database. In addition, I need information about which partition keys are used and how a table has been partitioned. How can I get a detailed report on my partitioned tables?

### **SOLUTION**

Oracle8 provides some new system tables that store information relating to partitioning. They contain information on what tables are partitioned, how many partitions exist for a table, which partition is associated with a tablespace, the number of rows in each of the partitions, and such. This information is quite useful for finding out which tables are partitioned and which are not.

#### **Generating a report on partitioned tables**

The following script shows you how to generate a listing of partitioned tables in a database (listing 5.30). It queries the DBA\_PART\_TABLES system table and retrieves information about the partitioned tables. The scripts can be modified to add more columns or you may change the format to suit your taste.

```
set echo off feedback off verify off;
set linesize 80 pagesize 60;

REM NAME           : mon_part_tables.sql
```

```

REM AUTHOR          : Noel.Y
REM USAGE           : Run from SQLPLUS
REM DESCRIPTION     : Generates a list of partitioned tables
REM REQUIREMENTS    : Must be run as SYS or Internal

col owner           format a10 heading "OWNER"
col table_name      format a15 heading "TABLE-NAME"
col partition_count format 999 heading "# of PARTITIONS"
col column_name     format a20 heading "COLUMN-NAME"
col TODAY          NEW_VALUE    _DATE

set termout off;
select to_char(SYSDATE,'fmMonth DD, YYYY') TODAY from DUAL;
set termout on;

TTITLE left _DATE CENTER "LIST OF TABLES PARTITIONED" Skip 1 -
CENTER "===== " skip 2

break on table_name on owner on partition_count;
spool &output_filename;
set heading on;
select
        T.table_name,
        T.owner,
        T.partition_count,
        K.column_name
from    dba_part_tables T,
        dba_part_key_columns K
where T.table_name = K.name
order by
        T.table_name;

clear columns;
spool off;
set feedback on verify on echo on;

```

**Listing 5.30 Script mon\_part\_tables.sql**

When the MON\_PART\_TABLES script is run, the report is generated as shown in listing 5.31. The output shows that there is only one partitioned table in the database. The table is PHONES which has four partitions and two columns in the partition key.

```

June 24, 1998          LIST OF TABLES PARTITIONED
                      =====
TABLE-NAME    OWNER    # of PARTITIONS    COLUMN-NAME
-----
PHONES        SYSTEM      4                  STATE_CODE
                                           ZIP_CODE
SQL>

```

**Listing 5.31 Output from script mon\_part\_tables.sql**

### **Generating a report on the partitioned table structure**

The MON\_PART\_TAB\_STRUCT.SQL script generates a report on the partitioned table structure (listing 5.32). It queries the DBA\_TAB\_PARTITION system table and retrieves information about how many partitions there are in a partitioned table, their respective tablespaces, the highest value assigned to a partition, and the number of actual rows in the partitioned tables. Again, the scripts can be modified to add more columns or the format changed to suit your taste.

```
set echo off feedback off verify off;
set long 18;
set linesize 80 pagesize 60;

REM NAME           : mon_part_tab_struct.sql
REM AUTHOR        : Noel.Y
REM USAGE         : Run from SQLPLUS
REM DESCRIPTION   : Generates a report on the partitioned
REM                Table structure.
REM REQUIREMENTS  : Must be run as SYS or Internal

col table_name      format a10      heading "TABLE"
col table_owner     format a10      heading "OWNER"
col partition_name  format a10      heading "PART-NAME"
col high_value      format 99999    heading "HIGHEST-VALUE"
col partition_position format 99999  heading "POSITION"
col tablespace_name format a10      heading "TABLESPACE"
col num_rows        format 99999    heading "# of ROWS"
col TODAY          NEW_VALUE        _DATE

set termout off;
select to_char(SYSDATE,'fmMonth DD, YYYY') TODAY from DUAL;
set termout on;

TTITLE left _DATE CENTER "PARTITIONED TABLES STRUCTURE" Skip 1 -
CENTER "===== " skip 2

break on table_name on table_owner;
spool &output_filename1
set heading on;
select  table_name,
        table_owner,
        partition_name,
        high_value,
        tablespace_name,
        num_rows
from
        dba_tab_partitions
order by table_name,
        partition_position;
clear columns;
spool off;
set feedback on verify on echo on;
```

**Listing 5.32** Script mon\_part\_tab\_struct.sql

When the MON\_PART\_TAB\_STRUCT script is run, the report shown in listing 5.33 is generated.

```

June 24, 1998                                PARTITIONED TABLES STRUCTURE
=====
TABLE      OWNER      PART-NAME  HIGHEST-VALUE  TABLESPACE  # of ROWS
-----
PHONES     SYSTEM     S1         'IA'           STATE1       1935
           S2         'MO'           STATE2       3417
           S3         'PA'           STATE3       25470
           S4         MAXVALUE      STATE4       12312

SQL>

```

**Listing 5.33** Output from script mon\_part\_tab\_struct.sql

---

**NOTE** If no row count appears in the “# of Rows” column, and you know that there *are* rows in the table, then you might have to run the ANALYZE command to update the system tables.

---

### CONCLUSION

The number of systems tables in Oracle8 has grown to include system tables that retain information about the partitioned tables. It stores information about the number of partitions in a table, their highest value associated with a partition, the partition key columns, and so forth. This information can be used to monitor the existing partitioned tables in a database.

## 5.7 What do I need to know about partition pruning?

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition pruning
<b>Operating system:</b> All	

### PROBLEM

What is partition pruning and how does it improve performance?

### SOLUTION

One of the greatest advantages of using this partition feature is not only manageability and availability but also increased performance. If a query requires only rows that are stored in a particular partition, then the Optimizer would scan only that partition, eliminating the others. It is referred to as partition pruning, or partition elimination.

Let us go through an example of how partition pruning occurs, as in the CUSTOMER table with the schema as shown in listing 5.34. It has CUST\_KEY as the primary key and has the CUST\_STATE as the partition key.

COLUMN	DATATYPE	COMMENTS
Cust_Key	Number	Primary key
Cust_Name	Varchar2(40)	
Cust_Add1	Varchar2(40)	
Cust_Add2	Varchar2(40)	
Cust_State	Varchar2(2)	Partition key
Cust_Zip	Number	
Cust_Balance	Number(7,2)	
Cust_Status	Char(2)	

**Listing 5.34 Customer table**

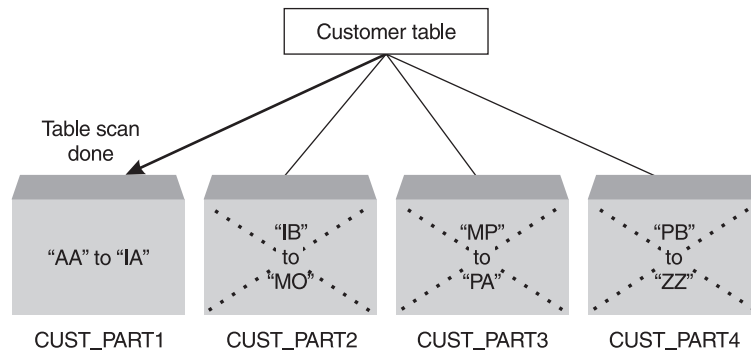
Let us assume that we have 4 partitions with the following range of values:

PARTITION	VALUES LESS THAN
CUST_PART1	"IA"
CUST_PART2	"MO"
CUST_PART3	"PA"
CUST_PART4	MAXVALUE

**Listing 5.35 Partition layout**

Now if we want to query the CUSTOMER table to find those customers who live in CA, Oracle would only have to look up one of the four partitions. This is because the Optimizer is partition-aware and knows what data goes into which partition. Therefore, it eliminates those that are not needed. This dramatically improves performance, especially for very large tables where a table scan of the entire table could involve considerable time. If we take a look at figure 5.9, we see that when a query retrieves rows only from a single partition, Oracle automatically will eliminate the other partitions.

```
Select cust_name, cust_balance
from customer
where cust_state = "CA";
```



**Figure 5.9 Partition elimination**

## CONCLUSION

Partition elimination occurs when the Optimizer only accesses the partitions that are needed to satisfy the query. In Oracle8 and Oracle8i, the Optimizer is partition-aware, and therefore knows what range of values lies in each of the partitions. So when a query is issued and the Optimizer knows which partition to access, it simply eliminates the other partitions and thereby improves performance.

## 5.8 *How can I create a local partitioned index in Oracle8?*

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Local partition index
<b>Operating system:</b> All	

### PROBLEM

Now that I have implemented a partitioned table, do I need to create a partitioned index? If so, how do I create a partitioned index?

### SOLUTION

In addition to table partitioning, Oracle supports partitioned indexes for non-clustered tables. Unlike a table partition, an index partition can have different types of range partitioning methods: either a local partitioned index or a global partitioned index. The local partitioned index can be either prefixed or nonprefixed, while the global partitioned index can only be a prefixed index. It is not necessary for a partitioned table to have a partitioned index. Both partitioned and nonpartitioned tables can have partitioned or nonpartitioned indexes.

Let us go through the different types of local partitioned indexes.

#### **Local partitioned indexes**

In the local index method, the index partition stores only those rows that relate to the underlying table partition. Oracle maintains the local index partitioning which is also called *equipartitioning*. A local index is created by using the LOCAL option in the CREATE INDEX command. Local partitioned indexes provide greater availability, manageability and performance. Oracle automatically keeps the index partition in sync with the table partitions.

A local partition can be either prefixed (unique or nonunique) or nonprefixed (unique only when the partition key is a subset of the index key or is nonunique).

#### **Local prefixed indexes**

If a local index is created on a column that is also used to partition the table and index, it is called a prefixed index. For example, if we have an EMPLOYEE table that has a table and index partitioned on the EMP\_NUMBER column, then it represents a local prefixed index, as shown in figure 5.10.

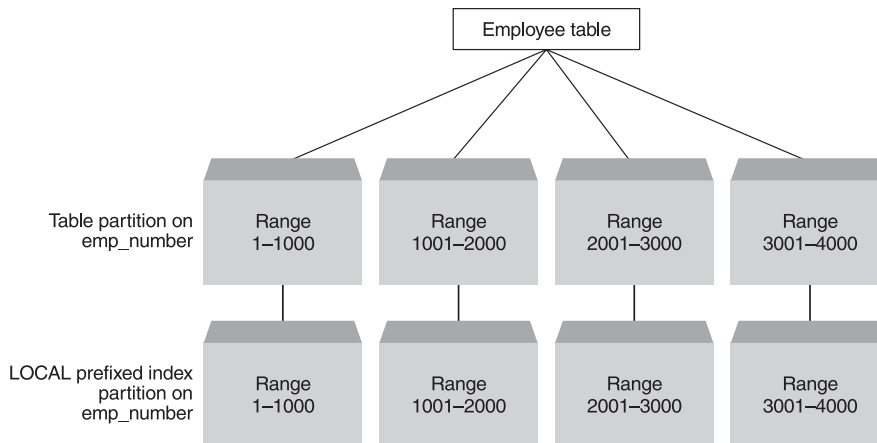
Before we proceed farther, let us describe the EMPLOYEE schema that will be used in the example.

```

CREATE TABLE employee
( emp_number    NUMBER,
  emp_name      CHAR(30),
  grade_level   NUMBER,
  dept_number   NUMBER,
  location_code CHAR(2),
  manager       NUMBER,
  salary        NUMBER(7,2),
  bonus         NUMBER(7,2))
PARTITION BY RANGE(emp_number)
(PARTITION EMP1
   VALUES LESS THAN (1001)
   TABLESPACE TBLSP1,
 PARTITION EMP2
   VALUES LESS THAN (2001)
   TABLESPACE TBLSP2,
 PARTITION EMP3
   VALUES LESS THAN (3001)
   TABLESPACE TBLSP3,
 PARTITION EMP4
   VALUES LESS THAN (MAXVALUE)
   TABLESPACE TBLSP4);

```

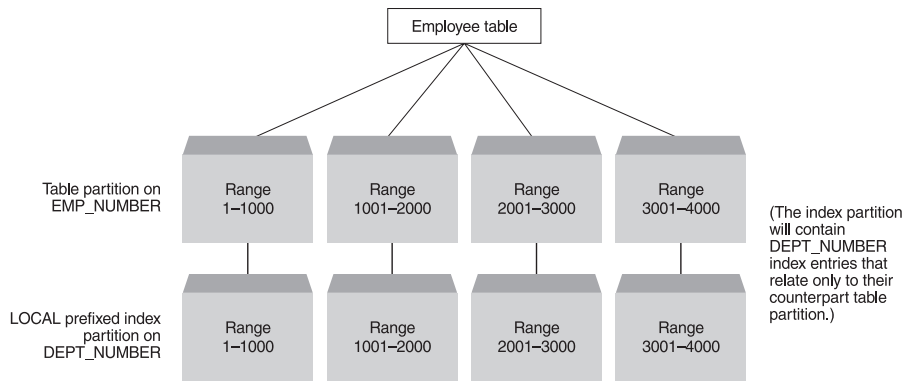
**Listing 5.36** The partitioned employee table



**Figure 5.10** Local prefixed index partition

### ***Local nonprefixed index***

A local nonprefixed index is one that is similar to the prefixed index except that the indexed column is not the same as the partitioned column. For example, if we have a CUSTOMER table that has been partitioned on the state column and the index has been created on the CUSTOMER\_NUMBER, then this would be called a nonprefixed index, as shown in figure 5.11.



**Figure 5.11 Local nonprefixed index partition**

The syntax to a local index is as follows:

```
CREATE INDEX <schema_name.index_name>
ON <schema_name.table_name>
(Column_name, ..... )
LOCAL
(PARTITION <partition_name1>
TABLESPACE <tablespace_name1>
[ STORAGE parameters ],
PARTITION <partition_name2>
TABLESPACE <tablespace_name2>
[ STORAGE parameters ], .... )
```

**Listing 5.37 Syntax for creating a local nonprefixed index**

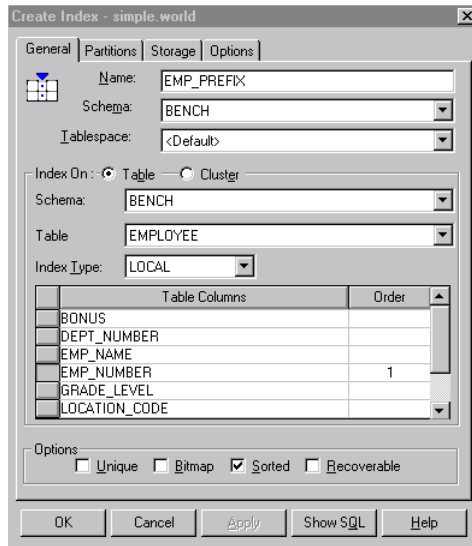
### STEPS TO CREATE A LOCAL PREFIXED INDEX

**Step 1.** When you create a local index, the number of partitions must be equal to the underlying table or you will receive Oracle error ORA-14024, “number of partitions of LOCAL index must equal that of the underlying table.” Listing 5.38 shows an example of how to create a local prefixed index on an employee table. This is a prefixed table since both the table and index are partitioned on the employee number.

```
SQL> CREATE INDEX emp_prefix
2 ON
3 employee (emp_number)
4 LOCAL
5 (PARTITION EMP1
6 TABLESPACE IDX_TBLSP1 ,
7 PARTITION EMP2
8 TABLESPACE IDX_TBLSP2 ,
9 PARTITION EMP3
10 TABLESPACE IDX_TBLSP3 ,
11 PARTITION EMP4
12 TABLESPACE IDX_TBLSP4 );
Index created.
SQL>
```

**Listing 5.38 Example of creating a local prefixed index**

**Step 2.** You can also create a local prefixed index using the OEM. Shown in figure 5.12 is an example of the Create Index dialog. Apart from the index name and user name, you will notice that it also allows you to select the Index Type, which can be either local or global. We will cover the global option in question 5.9.



**Figure 5.12**  
Create Index dialog

### **STEPS TO CREATE A LOCAL NONPREFIXED INDEX**

**Step 1.** In the example that follows, we create a local nonprefixed index on the EMPLOYEE table (listing 5.39). If you compare this example to the previous one, you will notice that the only difference is that now we are creating the index on a department number as opposed to the employee number.

```
SQL> CREATE INDEX emp_nonprefix
2   ON
3     employee (dept_number)
4   LOCAL
5     (PARTITION EMP1
6         TABLESPACE IDX_TBLSP1 ,
7         PARTITION EMP2
8         TABLESPACE IDX_TBLSP2 ,
9         PARTITION EMP3
10        TABLESPACE IDX_TBLSP3 ,
11        PARTITION EMP4
12        TABLESPACE IDX_TBLSP4 ) ;
Index created.
SQL>
```

**Listing 5.39** Example on creating LOCAL nonprefixed index

## CONCLUSION

In a local index method, the index partition stores only those entries that relate to the underlying table partition. A local partitioned index provides greater availability, manageability, and performance. Oracle automatically keeps the index partition in sync with the table partitions.

## 5.9 How do I create a global partitioned index in Oracle8?

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Global partition index
<b>Operating system:</b> All	

### PROBLEM

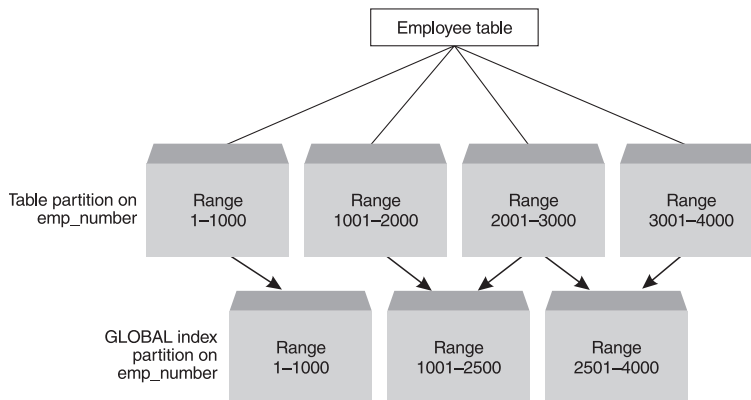
What is a global partitioned index and how does it differ from a local partitioned index? What are the advantages of a global partitioned index and when would you use it?

### SOLUTION

Global indexes have greater flexibility as to which partition key to use and are independent from the tables partitioning method. For example, you can have the EMPLOYEE table partitioned on the EMPLOYEE\_NUMBER with four partitions while having the global index on the EMPLOYEE\_NAME column with seven partitions. The number of partitions between the table and the index can differ in a global index but not in a local index.

In a global index, you can only have a prefixed index; nonprefixed indexes are not allowed, because they do not provide any manageability or performance gains. If you try to create a global partitioned index that is nonprefixed you will receive Oracle error message ORA-14038: “GLOBAL partitioned index must be prefixed.”

Figure 5.13, shows an example of a global index. Note that the number of partitions are different for the table and index partitions. Here the partition key is still the same EMP\_NUMBER for the table and index, making it a prefixed index.



**Figure 5.13**  
Global index  
partition

### **Limitations of a global index**

Although implementing a global index is similar to a local index, it has several limitations. First of all, global indexes are difficult to manage. If for some reason you drop, truncate, exchange, or split a table partition, the global partitioned index becomes useless. This is because the global index functions independently of the partitioned table. Let us go through an example of what happens when we actually drop a table partition.

#### **STEPS**

**Step 1.** First of all let us look at the explain plan (listing 5.40) for a simple select query that retrieves employee number 10. We already have an existing EMPLOYEE partitioned table with a global index, both of which have an employee number as the partition key.

```
SQL> explain plan for
  2  select * from employee
  3  where emp_number = 10;
Explained.
SQL> @plan
Query Plan
-----
SELECT STATEMENT      Cost = 1
  TABLE ACCESS BY GLOBAL INDEX ROWID EMPLOYEE
    INDEX RANGE SCAN EMP_GLOBAL
SQL>
```

#### **Listing 5.40 Explain plan for query accessing employee table**

**Step 2.** If we look at the explain plan, we find that the query will use the global index on an EMPLOYEE table. Once we know what the optimizer will be doing, we are then ready to run the query (listing 5.41).

```
SQL> select emp_number, emp_name
  2  from employee
  3  where emp_number = 10;
EMP_NUMBER EMP_NAME
-----
          10 Noel Y
SQL>
```

#### **Listing 5.41 Results from the query accessing employee table**

**Step 3.** The query retrieves one row from the EMPLOYEE table and everything appears to be in order. Now let us drop one of the partitions and see what the outcome will be (listing 5.42).

```
SQL> ALTER TABLE employee
  2  DROP PARTITION EMP2;
Table altered.
SQL>
```

#### **Listing 5.42 Command to drop partition EMP2**

The ALTER TABLE statement, with the DROP PARTITION option ran fine, as one would expect. And now for the big test: What happens if we rerun the same query?

```
SQL> select emp_number, emp_name
       2  from employee
       3  where emp_number = 10;
from employee
      *
ERROR at line 2:
ORA-01502: index 'BENCH.EMP_GLOBAL' or partition of such
index is in unusablestate
SQL>
```

**Listing 5.43 Output from running the same query again**

Well, the query fails, with Oracle error message ORA:01502, which means that a table partition is in an “unusable state” and therefore the global index is now useless. That does not mean that you cannot access the EMPLOYEE table at all. You can still do so via another index or a table scan.

The syntax for creating a global index is somewhat different than that for the local index (listing 5.44).

```
CREATE INDEX <schema_name.index_name>
       ON <schema_name.table_name>
GLOBAL PARTITION BY RANGE
      (Column_name, ..... )
      ( PARTITION <partition_name1>
        VALUES LESS THAN ( value_list)
          TABLESPACE <tablespace_name1>
          [STORAGE parameters ],
        PARTITION <partitoin_name2>
          VALUES LESS THAN ( values_list)
          TABLESPACE <tablespace_name2>
          [STORAGE parameters], ..... )
```

**Listing 5.44 Syntax for creating a global index**

**STEPS TO CREATE A GLOBAL INDEX**

**Step 1.** The following example (listing 5.45) shows how to create a global index:

```
SQL> CREATE INDEX emp_global
       2  ON
       3  employee (emp_number)
       4  GLOBAL PARTITION BY RANGE(emp_number)
       5  (PARTITION EMP1
       6  VALUES LESS THAN (1001)
       7  TABLESPACE IDX_TBLSP1,
       8  PARTITION EMP2
       9  VALUES LESS THAN (2501)
      10  TABLESPACE IDX_TBLSP2,
      11  PARTITION EMP3
      12  VALUES LESS THAN (MAXVALUE)
```

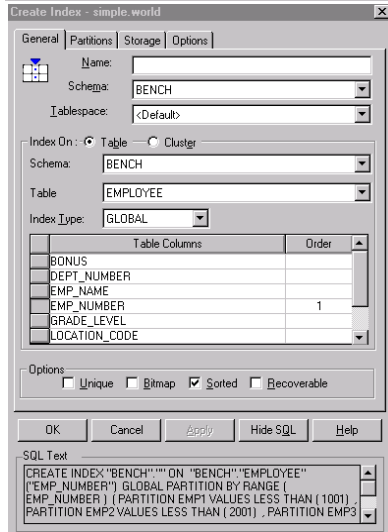
```

13                                     TABLESPACE IDX_TBLSP3 );
Index created.
SQL>

```

**Listing 5.45 An example of how to create a global index**

**Step 2.** You can also create a global index using the OEM. In figure 5.14, you can see a Create Index dialog with the Index Type set to GLOBAL.

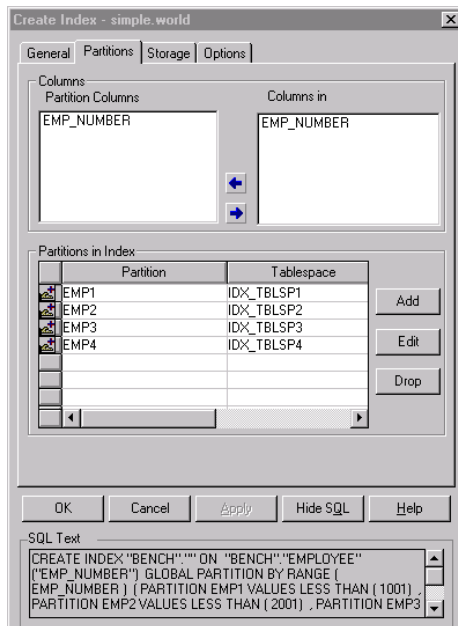


**Figure 5.14**  
**Create Index dialog**

In the Create Index dialog window you have a Partitions tab that allows you to set the number of partitions and their tablespaces. If you notice, in the Partition properties (figure 5.15), you can only choose one of the columns from the EMPLOYEE table. This is because Oracle8 only allows you to create a prefixed global index.

**CONCLUSION**

Global indexes have greater flexibility as to which partition key to use and they are independent from the table partitioning method. However global indexes are much more difficult to manage and do not offer availability. For a global index, you may only have a prefixed index; nonprefixed indexes are not allowed because they do not provide any manageability or performance gains.



**Figure 5.15**  
Index partition properties

## 5.10 How do I monitor a partitioned index in Oracle8?

<b>Applies to:</b> Oracle8 and Oracle8i	<b>CD Key:</b> Partition index
<b>Operating system:</b> ALL	

### PROBLEM

I would like to know what local and global indexes have been created on my database. I also need information about what type index has been built, and whether it is a prefixed or nonprefixed index. How can I obtain a detailed report on my partitioned indexes?

### SOLUTION

In Oracle8 and Oracle8i quite a few system tables are required to manage the partitions. You can query these system tables to derive more information on how a table has been partitioned. The `MON_PART_INDEXES.SQL` script (listing 5.46) queries the `DBA_PART_INDEXES` system table to generate a listing of partitioned indexes. The output from this script can be seen in listing 5.47. There is another script (listing 5.48) that generates a detailed report on the partitioned index. The output from this script is shown in listing 5.49.

```
set echo off feedback off verify off;
set linesize 80 pagesize 60;

REM NAME           : mon_part_indexes.sql
REM AUTHOR          : Noel.Y
REM USAGE           : Run from SQLPLUS
```

```

REM DESCRIPTION   : Generates a list of partitioned Indexes
REM REQUIREMENTS  : Must be run as SYS or Internal

col owner          format a10 heading "OWNER"
col table_name     format a15 heading "TABLE-NAME"
col index_name     format a15 heading "INDEX-NAME"
col partition_count format 999 heading "# of PARTITIONS"
col column_name    format a20 heading "COLUMN-NAME"
col TODAY          NEW_VALUE      _DATE

set termout off;
select to_char(SYSDATE,'fmMonth DD, YYYY') TODAY from DUAL;
set termout on;

TTITLE left _DATE CENTER "LIST OF PARTITIONED INDEXES" Skip 1 -
CENTER "===== " skip 2

break on owner on table_name on index_name;
spool &output_filename;
set heading on;
select
      I.owner,
      K.table_name,
      I.index_name,
      I.partition_count,
      K.column_name
from    dba_part_indexes I,
      dba_ind_columns K
where I.index_name = K.index_name
      order by
I.owner,K.table_name,I.index_name;
clear columns;
spool off;
set feedback on verify on echo on;

```

**Listing 5.46 Script mon\_part\_indexes.sql**

```

SQL> @mon_part_indexes
June 27, 1998                LIST OF PARTITIONED INDEXES
=====
OWNER      TABLE-NAME      INDEX-NAME      # of PARTITIONS  COLUMN-NAME
-----
BENCH      EMPLOYEE         EMP_NONPREFIX      4      DEPT_NUMBER
                        EMP_PREFIX         4      EMP_NUMBER
SQL>

```

**Listing 5.47 Output from script mon\_part\_indexes.sql**

```

set echo off feedback off verify off;
set long 18;
set linesize 80 pagesize 60;

REM NAME          : mon_part_ind_struct.sql
REM AUTHOR        : Noel.Y
REM USAGE         : Run from SQLPLUS
REM DESCRIPTION   : Generates a report on the partitioned
REM               Index structure.

```

```

REM REQUIREMENTS : Must be run as SYS or Internal

col table_name          format a10          heading "TABLE"
col table_owner         format a10          heading "OWNER"
col partition_name     format a10          heading "PART-NAME"
col high_value         format 99999        heading "HIGHEST-VALUE"
col partition_position format 99999        heading "POSITION"
col tablespace_name    format a10          heading "TABLESPACE"
col num_rows           format 99999        heading "# of ROWS"
col TODAY              NEW_VALUE          _DATE

set termout off;
select to_char(SYSDATE,'fmMonth DD, YYYY') TODAY from DUAL;
set termout on;

TTITLE left _DATE CENTER "PARTITIONED INDEXES STRUCTURE" Skip 1 -
CENTER "===== " skip 2
break on table_name on index_name;
spool &output_filename;
set heading on;
select K.table_name,
       I.index_name,
       I.partition_name,
       I.high_value,
       I.tablespace_name,
       I.num_rows

from   dba_ind_partitions I,
       dba_ind_columns K

where  I.index_name = K.index_name
order by I.index_name,
         partition_position;
clear columns;
spool off;
set feedback on verify on echo on;

```

**Listing 5.48 Script mon\_part\_ind\_struct.sql**

```

SQL> @mon_part_ind_struct
SQL> set echo off
June 27, 1998

```

PARTITIONED INDEXES STRUCTURE					
=====					
TABLE	INDEX-NAME	PART-NAME	HIGHEST-VALUE	TABLESPACE	# of ROWS
-----					
EMPLOYEE	EMP_NONPREFIX	EMP1	1001	IDX_TBLSP1	125
		EMP2	2001	IDX_TBLSP2	101
		EMP3	3001	IDX_TBLSP3	168
		EMP4	MAXVALUE	IDX_TBLSP4	124
	EMP_PREFIX	EMP1	1001	IDX_TBLSP1	125
		EMP2	2001	IDX_TBLSP2	101
		EMP3	3001	IDX_TBLSP3	168
		EMP4	MAXVALUE	IDX_TBLSP4	124

```

SQL>

```

**Listing 5.49 Output from script mon\_part\_ind\_struct.sql**

## CONCLUSION

Oracle8 provides you with system tables that have all of the details necessary to manage partitioning. To retrieve information about your partitioned index, you can query the DBA\_PART\_INDEXES, DBA\_PART\_COLUMNS, DBA\_IND\_PARTITIONS, and DBA\_IND\_COLUMNS system table. You can also run the MON\_PART\_INDEXES.SQL script to generate a report on the partitioned indexes in a database.

## 5.11 How do I create a hash-partitioned table?

8i	Applies to: Oracle8i	CD Key: Hash partition table
	Operating system: All	

### PROBLEM

I would like to partition my large table using a simple partitioning method that will evenly distribute the data across the partitions and among the tablespaces. How do I create a hash partitioned table?

### SOLUTION

Hash partitioning is a new method of partitioning a table using a hash function on the partitioned columns. Rows are mapped to the partitions based on the hash value of the partitioning key. They are fairly easy to create and manage. Basically, the database automatically distributes the data among the hash partitions using the Hash function. This improves both the query performance and also that of parallel DML by spreading the I/O across multiple tablespaces.

Hash partitioning is useful if you do not know how much of the data will be in a range. The number of partitions should be to a power of two such as 2, 4, 8, to achieve a given distribution of the data. Local indexes on hash partitioned tables are equipartitioned with the table data.

Hash partitions may be stored in any specified tablespaces, which gives you complete flexibility (see listing 5.50). The local indexes on a hash partition are equipartitioned with the table's data.

```
CREATE TABLE ....  
PARTITION BY HASH ( <column_list> )  
PARTITIONS (Quantity) STORE IN (tablespace_name, ) **Method 1  
PARTITION (partition_name) (tablespace_name) ** Method 2
```

#### Listing 5.50 Create Table... for hash partitioned table

The column\_list is a listing of columns used to determine the hash partition to store the row. You cannot specify more than 16 columns and it cannot contain ROWID or UROWID columns.

The hash partitioning can be specified in two ways:

**1 *Specifying the number of partitions***

In this method, you specify the number of partitions and the tablespace names. The STORE IN clause in the CREATE TABLE command specifies the tablespaces to be used. The number of tablespaces does not have to equal the number of partitions. If the number of partitions is greater than the tablespaces specified, Oracle will assign the additional partitions in a round-robin manner.

**2 *Specifying individual partitions***

In this method, you have to specify the individual partitions by name. The TABLESPACE clause is used to specify where the partition will be stored (listing 5.51).

```
CREATE TABLE <table_name>
  STORAGE (Initial xx M)
  PARTITION BY HASH (Column_list)
    (PARTITION <part-1> TABLESPACE <tbl-space-1>
      PARTITION <part-2> TABLESPACE <tbl-space-2>
      PARTITION <part-3> TABLESPACE <tbl-space-3>);

CREATE TABLE videos
  STORAGE (INITIAL 1M)
  PARTITION BY HASH (video_number)
    (PARTITION p1 TABLESPACE t1,
     PARTITION p2 TABLESPACE t2);
```

**Listing 5.51 Example of creating a hash-partitioned table**

As discussed previously, there are two methods that can be used to create a hash-partitioned table—one by specifying the number of partitions and the other by specifying each individual partition. Now let us walk through an example of how to create a hash partitioned table in Oracle8i using each of the methods.

**Method 1—*Specifying the number of partitions.*** The example shown in listing 5.52 shows you how to create a hash partitioned table. It has 8 partitions that will be stored in three tablespaces named T1, T2, and T3. The table is partitioned using the EMP\_NUMBER column.

```
SVRMGR> CREATE TABLE employee
  2>      ( emp_number          NUMBER PRIMARY KEY,
  3>        emp_name            CHAR(40),
  4>        emp_address_1       CHAR(40),
  5>        emp_address_2       CHAR(40),
  6>        emp_city            CHAR(20),
  7>        emp_state           CHAR(2),
  8>        emp_dept            CHAR(2),
  9>        emp_salary          NUMBER,
 10>        emp_bonus           NUMBER,
 11>        emp_hire_date        DATE )
 12>      STORAGE (INITIAL 1M NEXT 1M),
```

```

13>          PARTITION BY HASH (emp_number) PARTITIONS 8
14>          STORE IN ( T1, T2, T3 );
Statement processed.
SVRMGR>

```

**Listing 5.52 Example of creating a hash partitioned table**

**Method 2—Specifying the individual partitions.** You can also create a hash partitioned table by specifying each individual partition for greater granularity, which gives you more control. In the example shown in listing 5.53, you can see that we are creating a hash partitioned table called EMPLOYEE that is partitioned using the EMP\_NUMBER column. There are only three partitions, with each partition on a separate tablespace, so partition P1 is on tablespace T1, partition P2 on tablespace T2, and so on.

```

SVRMGR> CREATE TABLE employee
2>          ( emp_number          NUMBER PRIMARY KEY,
3>            emp_name            CHAR(40),
4>            emp_address_1      CHAR(40),
5>            emp_address_2      CHAR(40),
6>            emp_city           CHAR(20),
7>            emp_state          CHAR(2),
8>            emp_dept           CHAR(2),
9>            emp_salary         NUMBER,
10>           emp_bonus          NUMBER,
11>           emp_hire_date       DATE )
12>          STORAGE ( INITIAL 1M NEXT 1M),
13>          PARTITION BY HASH (emp_number)
14>          (PARTITION P1 TABLESPACE T1,
15>           PARTITION P2 TABLESPACE T2,
16>           PARTITION P3 TABLESPACE T3 );
Statement processed.
SVRMGR>

```

**Listing 5.53 Example of creating a hash partitioned table**

**STEPS**

**Step 1—Insert data into hash partitions.** Let us now insert some rows into the hash partitioned table. As you can see from listing 5.54, the inserts are regular INSERT statements. We assume that the NLS date format is DD-MON-YYYY. Once the insert statement has been executed, make certain that you also COMMIT the data.

```

SVRMGR> INSERT INTO employee values
2>          (100,'John D','1250 E.North Street','Mail Stop 31',
3>          'Microsoft Way','TX','18',5000.00,1200.00
4>          '01-JAN-1999'
1 row processed.
SVRMGR> INSERT INTO employee values
2>          (200,'Noel T','880 South 23 Street','Mail Stop 11',
1 row processed.
SVRMGR>

```

**Listing 5.54 Inserting rows into the partitioned table**

**Step 2—Querying the table.** Once the data has been inserted into the hash partitioned table, you may begin querying the table. As you can see from listing 5.55, if we query the EMPLOYEE table, two rows are returned to us. Now, take a look at listing 5.56, where we query the same EMPLOYEE table but specify PARTITION (partition\_name); as you can see for partition P1, we get back only one row, for partition P2 there are no rows, and for partition P3 there is one row. Unlike range partitioning where you have to specify the range definition (i.e., the lower and upper limit) for each of the partitions, in hash partitioning, the data is distributed automatically among the partitions.

```
SVRMGR> select emp_number,emp_name
        2> from employee;
EMP_NUMBER EMP_NAME
-----
          200 Noel T
          100 John D
2 rows selected.
SVRMGR>
```

**Listing 5.55 Querying the employee table**

```
SVRMGR> select emp_name from employee partition (p1);
EMP_NAME
-----
Noel T
1 row selected.
SVRMGR> select emp_name from employee partition (p2);
EMP_NAME
-----
0 rows selected.
SVRMGR> select emp_name from employee partition (p3);
EMP_NAME
-----
John D
1 row selected.
SVRMGR>
```

**Listing 5.56 Querying the employee table for a specific partition**

With hash partitioning, like range partitioning, you can SPLIT, MERGE, DROP, COALESCE and ADD partitions.

The statements to manage the hash partitions are:

- ALTER TABLE ... SPLIT PARTITION
- ALTER TABLE ... DROP PARTITION
- ALTER TABLE ... MERGE PARTITION
- ALTER TABLE ... COALESCE PARTITION
- ALTER TABLE ... ADD PARTITION

The COALESCE PARTITION option removes a single hash partition and redistributes the data. The ADD PARTITION option adds a single hash partition and redistributes the data. You can also add the hash partition in parallel.

## CONCLUSION

Hash partitioning is a new method of partitioning a table using the hash function on the partitioned columns. It is useful if you do not know how much data will be in a range. Rows are mapped to the partitions based on the hash value of the partitioning key. Under hash partitioning, the data is distributed using the hash function and stores it in a distributed fashion, therefore spreading the I/O across multiple tablespaces. This improves query performance and also that of parallel DML.

## 5.12 How do I create a composite-partitioned table?

8j	<b>Applies to:</b> Oracle8i	<b>CD Key:</b> Composite partition table
	<b>Operating system:</b> All	

### PROBLEM

I would like to create a table based on the new composite partitioning method. What procedures should I follow to create a composite partitioned table?

### SOLUTION

Composite partitioning uses a combination of the range partition and the hash partition (listing 5.57). It first partitions the data using the range method, and within each partition it uses the hash method. It provides the manageability and availability of range partitioning with the data distribution advantages of hash partitioning. You must specify the ranges of values for the primary partitions of the table and then specify the number of hash subpartitions.

Composite partitioning is suitable for both historical data and for distributing data evenly across tablespaces.

Characteristics of a composite partitioned table are:

- Provides the data placement advantage of range partitioning
- Provides the parallelism advantage of hash partitioning
- Can name subpartitions
- Can store subpartitions in specific tablespaces
- Can build local indexes
- Can build range-partitioned global indexes
- Can name index subpartitions
- Can store index subpartitions in specific tablespaces

```
PARTITION BY RANGE ( <column_list> )
    SUBPARTITION BY HASH ( <column_list>, )
    SUBPARTITIONS (Quantity) STORE IN (Tablespace_name)
```

#### Listing 5.57 Syntax for creating a composite partitioned table

The `subpartition_clause` specifies that the table should be subpartitioned by hash. The subpartitioning `column_list` is unrelated to the partitioning key. The

quantity specifies the number of subsections for each partition. The default subpartition is 1. You can specify up to 63,999 partitions and 63,999 sub-partitions.

## STEPS

**Step 1.** Let us create a new composite partitioned table by specifying both the range and hash. In the example shown in listing 5.58, the CUSTOMER table is first range-partitioned by STATE\_CODE. As you can see, there are four range partitions P1, P2, P3, and P4. Now each of those partitions is further subpartitioned into six subsections. Therefore a total of 24 partitions exists on the CUSTOMER table—one subpartition from each of the partitions would be in each tablespace.

```
SVRMGR> CREATE TABLE customer
2>      ( cust_number  NUMBER PRIMARY KEY,
3>        name         CHAR(40),
4>        address_1    CHAR(40),
5>        address_2    CHAR(40),
6>        state_code   CHAR(2),
7>        zip_code     NUMBER)
8>      PARTITION BY RANGE (state_code)
9>        SUBPARTITION BY HASH(cust_number) SUBPARTITIONS 6
10>       STORE IN (T1, T2, T3)
11>      ( PARTITION P1 VALUES LESS THAN ('IA'),
12>        PARTITION P2 VALUES LESS THAN ('MO'),
13>        PARTITION P3 VALUES LESS THAN ('PA'),
14>        PARTITION P4 VALUES LESS THAN (MAXVALUE) );
Statement processed.
SVRMGR>
```

### Listing 5.58 Syntax for creating a composite partitioned table

To manage composite subpartitions, Oracle provides you with the following:

- ALTER TABLE ... MODIFY SUBPARTITION.....
- ALTER TABLE ... REBUILD SUBPARTITION....
- ALTER TABLE ... EXCHANGE SUBPARTITION.....
- ALTER TABLE ... ADD SUBPARTITION ....
- ALTER TABLE ... MODIFY SUBPARTITION ....
- ALTER TABLE ... TRUNCATE SUBPARTITION .....

Besides subpartitions, you can do the same with partitions.

- ALTER TABLE ... ADD PARTITION ....
- ALTER TABLE ... COALESCE PARTITION
- ALTER TABLE ... DROP PARTITION
- ALTER TABLE ... EXCHANGE PARTITION
- ALTER TABLE ... MERGE PARTITION
- ALTER TABLE ... MOVE PARTITION ...
- ALTER TABLE ... REBUILD PARTITION
- ALTER TABLE ... SPLIT PARTITION ...

## CONCLUSION

Composite partitioning uses a combination of the range partition and the hash partition. Under this method, Oracle first partitions the data using the range method and within each partition it uses the hash method. This method offers the improved manageability and availability of range partitioning, together with the data distribution advantages of hash partitioning. Composite partitioning is suitable both for historical data and for distributing data evenly across tablespaces.

## 5.13 How do I merge partitions?

<b>8i</b>	<b>Applies to:</b> Oracle8i	<b>CD Key:</b> Merge partition
	<b>Operating system:</b> All	

### PROBLEM

I have been using a range-partitioned table for a large table. It has over 24 partitions and the data does not seem to be distributed evenly across partitions. I would like to merge partitions that have less data so that I can manage them more easily. How do I merge partitions?

### SOLUTION

The inverse of splitting partitions is to merge them. This is a new feature in Oracle8i. You can only merge two adjacent partitions of a range or composite partitioned table into a single partition. The syntax for merging two partitions is shown in listing 5.59.

```
ALTER TABLE <table_name>
MERGE PARTITION <partition-name-1>, <partition-name-2>
INTO PARTITION <new_partition_name>
```

#### Listing 5.59 Merge partition syntax

To merge partitions, the table cannot be hash partitioned—only range or composite range are allowed. You will get an ORA-14255 error if you try to merge partitions on a hash-partitioned table.

### STEPS

First of all let us assume that we have a CUSTOMER table that is range partitioned. Listing 5.60 shows the CUSTOMER table definition.

```
SVRMGR> create table customer
2>           ( cust_number  NUMBER PRIMARY KEY,
3>           name           CHAR(40),
4>           address_1      CHAR(40),
5>           address_2      CHAR(40),
6>           state_code     CHAR(2),
7>           zip_code       NUMBER )
8>           PARTITION BY RANGE(state_code)
9>           (PARTITION S1 VALUES LESS THAN ('IA')
10>           TABLESPACE T1
```

```

11>                STORAGE (INITIAL 1M NEXT 1M) ,
12>                PARTITION S2 VALUES LESS THAN ('MO')
13>                TABLESPACE T2
14>                STORAGE (INITIAL 1M NEXT 1M) ,
15>                PARTITION S3 VALUES LESS THAN ('PA')
16>                TABLESPACE T3
17>                STORAGE (INITIAL 1M NEXT 1M) ,
18>                PARTITION S4 VALUES LESS THAN (MAXVALUE)
19>                TABLESPACE T4)
20>                STORAGE (INITIAL 1M NEXT 1M) ;
Statement processed.
SVRMGR>

```

**Listing 5.60 Creating a range-based partition table**

**Step 1.** Let us add five rows to the CUSTOMER table, with various state codes as shown in listing 5.61. This will allow the rows to be distributed over the partitions, since the partition key is the state code.

```

SVRMGR> insert into customer values (1001,'JOHN',
2>      '1260 New Memory Ave','APT 901T1','CA',95691);
1 row processed.
SVRMGR> insert into customer values (1002,'NOEL',
2>      '180 Old Storage Rd',' ','TX',89691);
1 row processed.
SVRMGR> insert into customer values (1003,'PARAG',
2>      '70 Disney Rd',' ','IN',89691);
1 row processed.
SVRMGR> insert into customer values (1004,'RONALD',
2>      '1234 Gold Street',' ','CA',89691);
1 row processed.
SVRMGR> insert into customer values (1005,'MANMEET',
2>      '349 12 Street',' ','NY',89691);
1 row processed.
SVRMGR>

```

**Listing 5.61 Inserting rows into customer table**

On querying the CUSTOMER table, we can see that there are five rows in the table (listing 5.62).

```

SVRMGR> select name, state_code
2>      from customer;
NAME                                ST
-----
JOHN                                CA
RONALD                              CA
PARAG                                IN
MANMEET                              NY
NOEL                                  TX
5 rows selected.
SVRMGR>

```

**Listing 5.62 Querying the customer table**

**Step 2.** On querying the customer table and specifying the partition name, we obtain the results as shown in listing 5.63. As you can see, there are one or two rows in each partition.

```
SVRMGR> select name, state_code from customer partition (s1);
NAME                                ST
-----
JOHN                                CA
RONALD                              CA
2 rows selected.
SVRMGR> select name, state_code from customer partition (s2);
NAME                                ST
-----
PARAG                                IN
1 row selected.
SVRMGR> select name, state_code from customer partition (s3);
NAME                                ST
-----
MANMEET                              NY
1 row selected.
SVRMGR> select name, state_code from customer partition (s4);
NAME                                ST
-----
NOEL                                  TX
1 row selected.
SVRMGR>
```

**Listing 5.63 Querying the customer table for a specific partition**

**Step 3.** Now let us try to merge partitions S3 and S4, for example, calling the new partition S34, and attempt to store it in tablespace T5 as shown in listing 5.64.

```
SVRMGR> ALTER TABLE customer
2>          MERGE PARTITIONS S3, S4
3>          INTO PARTITION S34
4>          TABLESPACE T5;
Statement processed.
SVRMGR>
```

**Listing 5.64 Merging partitions**

**Step 4.** On querying partition S3, we get error ORA-02149, which relates to a nonexistent partition, as shown in listing 5.65.

```
SVRMGR> select name, state_code from customer partition (s3);
select name, state_code from customer partition (s3)
*
ORA-02149: Specified partition does not exist
```

**Listing 5.65 Querying the customer table for deleted partitions**

**Step 5.** Finally, we query the new partition S34 and obtain the two rows that were originally in partition S3 and S4 (listing 5.66).

```

SVRMGR> select name, state_code from customer partition (s34);
NAME                                     ST
-----
MANMEET                                  NY
NOEL                                      TX
2 rows selected.
SVRMGR>

```

**Listing 5.66 Querying the customer table for the new partition**

### CONCLUSION

Starting with Oracle8*i*, you can MERGE partitions into a single partition. However, you can only merge two *adjacent* partitions of a range or composite partitioned table. This feature is very important if you have too many partitions and would like to consolidate some for better manageability and performance.

## 5.14 How do I enable row movement between partitions?

<b>8<i>i</i></b>	<b>Applies to:</b> Oracle8 <i>i</i>	<b>CD Key:</b> Row movement
	<b>Operating system:</b> All	

### PROBLEM

I have an application that changes the partition key value for a partitioned table. The update to the partition key fails with error message ORA-14402, “updating partition key column would cause a partition change.” How might I enable row movement between partitions?

### SOLUTION

In Oracle8*i* you can enable or disable row movement between partitions. The syntax for enabling such a movement is found in the ALTER TABLE statement with the ENABLE/DISABLE ROW MOVEMENT option (listing 5.67). If you are using hash partitioning and your application changes the partition key such that it will change the partition, then Oracle will generate an error. To overcome this problem, version 8*i* allows you to enable row movement between partitions with some overheads.

```

ALTER TABLE <table_name>
    ENABLE/DISABLE ROW MOVEMENT

```

**Listing 5.67 Syntax to enable/disable row movement**

### STEPS

**Step 1.** Let us assume that we have a CUSTOMER table that is range-based partitioned on STATE\_CODE. Listing 5.68 shows two rows being inserted into the table.

```

SVRMGR> insert into customer values (1001,'JOHN',
2>      '1260 New Memory Ave', 'APT 901T1', 'CA', 95691);
1 row processed.
SVRMGR> insert into customer values (1002,'NOEL',

```

```

2>          '880 Old Storage Rd',' ','TX',89691);
1 row processed.
SVRMGR> commit;
Statement processed.
SVRMGR>

```

**Listing 5.68 Inserting rows into the customer table**

**Step 2.** Now if we try to update the CUSTOMER table such that the STATE\_CODE value changes and the row has to migrate to another partition, the operation will fail with ORA-14002 as shown in listing 5.69.

```

SVRMGR> update customer set state_code = 'NY'
2> where cust_number = 1001;
ORA-14402: updating partition key column would cause a partition change
SVRMGR>

```

**Listing 5.69 Updating the customer table**

**Step 3.** To allow ROW MOVEMENT for that table you would have to simply use the ALTER TABLE statement or the CREATE TABLE statement to enable it. Listing 5.70 shows you how to enable the ROW MOVEMENT of the CUSTOMER table.

```

SVRMGR> ALTER TABLE customer
2> ENABLE ROW MOVEMENT;
Statement processed.
SVRMGR>

```

**Listing 5.70 Using the ALTER command to enable ROW MOVEMENT**

**Step 4.** Once the table has been enabled, you can then use the UPDATE command to change the STATE\_CODE column for the CUSTOMER table. As you can see from listing 5.71, the update was successful.

```

SVRMGR> update customer set state_code = 'NY'
2> where cust_number = 1001;
1 row processed.
SVRMGR> commit;
Statement processed.
SVRMGR>

```

**Listing 5.71 Updating the customer table**

**CONCLUSION**

In Oracle8i you can enable or disable row movement between partitions using the ALTER TABLE statement with the ROW MOVEMENT option. Previously, if you were using a range-partitioned table and your application updated the partition key such that it would change the partition, Oracle would generate an error and fail to execute the command. However, with Oracle8i, you can now enable or disable rows to move between partitions.