7. Managing Data – Tables, Indexes and Constraints.

Abstract: Previous lesson have discussed Oracle's architecture: physical and logical structures of the database. Data is stored in Oracle as rows and columns. This lesson covers the options available when creating tables, shows how to quickly retrieve data by using indexes, and discusses how the Oracle database can enforce business rules by using integrity constraints.

Contents

Objective: After completing this lesson the students will be able to:

- Identify the various methods of storing dataObjective 2
- Describe Oracle datatypes
- Distinguish between an extended versus a restricted ROWID
- Describe the structure of a row
- Create regular and temporary tables
- Manage storage structures within a table
- Reorganize, truncate, drop a table
- Drop a column within a table
- List different types of indexes and their uses
- Create various types of indexes
- Reorganize indexes
- Drop indexes

- Get index information from the data dictionary
- Monitor the usage of an index
- Implement data integrity constraints
- Maintain integrity constraints
- Obtain constraint information from the data dictionary

1. Storing Data

A table is the basic form of data storage in Oracle. You can think of a table as a spreadsheet having column headings and many rows of information. A schema, or a user, in the database owns the table. The table columns have a defined datatype — the data stored in the columns should satisfy the characteristics of the column. You can also define rules for storing data in the columns using integrity constraints. Oracle9i has various types of tables to suit your data storage needs. A table by default means a relational permanent table. The following types of tables are available in Oracle9i to store data.

Relational Simply known as a table, the relational table is the most common method for storing data. These tables are permanent and can be partitioned. When you partition a table, you break it into multiple smaller pieces, which improves performance and makes the table easier to manage. To create a relational table, you use the CREATE TABLE ... ORGANIZATION HEAP statement. Since ORGANIZATION HEAP is the default, it can be omitted.

Temporary Temporary tables store private data or data that is specific to a session. Other users in the database cannot use these data. Temporary tables are used for temporary data manipulation or for storing intermediary results. To create a temporary table, you use the CREATE GLOBAL TEMPORARY TABLE statement.

Index Organized Index Organized Tables (IOTs) store the data in a structured primary key sorted manner. You must define a primary key for each IOT. These tables are similar to relational tables that have a primary key, but they do not use separate storage for the table and primary key as relational tables do. To create an IOT, you use the CREATE TABLE ... ORGANIZATION INDEX statement.

External The external table type was introduced in Oracle9i. As the name indicates, data is stored outside the Oracle database in flat files. External tables are read-only, and no indexes are allowed on external tables. Column names defined in the Oracle database are mapped to the columns in the external file. The default driver used to read external table is SQL*Loader. To create an external table, you use the CREATE TABLE ... ORGANIZATION EXTERNAL statement.

Object Object tables are a special kind of tables that support the object-oriented features of the Oracle9i database. In an object table, each row represents an object. We have already discussed in the previous lessons the logical storage structures and parameters. Let's see how these structures can be related to a table. The following sections discuss how to create and manipulate the types of tables in Oracle9i.

1.1. Creating Tables

Objective: Create regular and temporary tables

To create a table, you use the CREATE TABLE command. You can create a table under the username used to connect to the database, or, with proper privileges, you can create a table under another username. A database user can be referred to as a schema or as an owner, when the user owns objects in the database. The simplest form of creating a table is as follows:

```
CREATE TABLE ORDERS (
ORDER_NUM NUMBER,
ORDER_DATE DATE,
PRODUCT_CD VARCHAR2 (10),
QUANTITY NUMBER (10,3),
STATUS CHAR);
```

ORDERS is the table name; the columns in the table are specified in parentheses separated by commas. The table is created under the username used to connect to the database; to create the table under another schema, you need to qualify the table with the schema name. For example, if you want to create the ORDERS table as being owned by SCOTT, create the table by using CREATE TABLE SCOTT.ORDERS ( ).

A column name and a datatype identify each column. For certain datatypes, you can specify a maximum width. You can specify any Oracle built-in datatype or user-defined datatype for the column definition. When specifying user-defined datatypes, the user-defined type must exist before creating the table.

Objective: Describe Oracle datatypes.

Oracle9i has three categories of built-in datatypes: scalar, collection, and relationship. Collection and relationship datatypes are used for object-relational functionality of Oracle9i. Table 1 lists the built-in scalar datatypes in Oracle.

Table 1. Oracle Built-in Scalar Datatypes

Datatype	**Description**
CHAR (<size> [BYTE | CHAR])	Fixed-length character data with length specified inside parentheses. Data is space padded to fit the column width. You can also include the optional keywords BYTE and CHAR inside

	parentheses along with size to indicate if the size specified is in bytes or in characters. BYTE is the default. Size defaults to 1 byte if not defined. Maximum is 2000 bytes.
VARCHAR (<size> [BYTE | CHAR])	Same as VARCHAR2.
VARCHAR2 (<size> [BYTE | CHAR])	Variable-length character data. Maximum allowed length is specified in parentheses. You must specify a size; there is no default value. Maximum is 4000 bytes. Unlike the CHAR datatype, VARCHAR2 columns are not blank padded with trailing spaces if the column value is shorter than its maximum specified length. You can specify the size in bytes or characters; by default the size is in characters.
NCHAR (<size>)	Similar to CHAR, but used to store Unicode character set data. NCHAR datatype is fixed length, maximum size 2000 bytes, and default size 1 character.
NVARCHAR2 (<size>)	Same as VARCHAR2; stores Unicode variable length data. The size is specified in characters, and the maximum allowed size is 4000 bytes.
LONG	Stores variable-length character data up to 2GB. Use CLOB or NCLOB datatypes instead. Provided since Oracle9i for backward compatibility. Can have only one LONG column per table.
NUMBER (<precision>, <scale>)	Stores fixed and floating-point numbers. You can optionally specify a precision (total length including decimals) and scale (digits after decimal point). The default is 38 digits of precision, and the valid range is between –1 × 10−130 and 9.999 99125.
DATE	Stores date data. Has century, year, month, date, hour, minute, and seconds internally. Can be displayed in various formats. You can store the dates from January 1, 4712 BC to December 31, 9999 AD. If you specify a date value without the time component, the default time is 12AM (midnight 00:00:00 hrs).
TIMESTAMP [(<precision>)]	TIMESTAMP datatype stores date and time information with fractional seconds precision. The only difference between DATE and TIMESTAMP datatypes is the ability to store fractional seconds up to a precision of 9 digits. The default precision is 6 and can range from 0 to 9.
TIMESTAMP [(<precision>)]	TIMESTAMP WITH TIME ZONE is similar to the TIMESTAMP datatype, but stores the time zone displacement. Displacement is the difference between the local time and the Universal Time

WITH TIME ZONE	Coordinate (UTC), also known as Greenwich Mean Time. The displacement is represented in hours and minutes.
TIMESTAMP [(<precision>)] WITH LOCAL TIME ZONE	TIMESTAMP WITH LOCAL TIME ZONE is similar to the TIMESTAMP datatype, but includes the time zone displacement. TIMESTAMP WITH LOCAL TIME ZONE does not store the displacement information in the database, but stores the time as a normalized form of database time zone. The data is always stored in the database time zone, but when the user retrieves data, it is shown in the users local session time zone.
INTERVAL YEAR [(precision)] TO MONTH	Used to represent a period of time as years and months. The precision specifies the precision needed for the year field, and its default is 2. The precision can have values from 0 to 9. This datatype can be used to store the difference between two date time values, in which the only significant portions are the year and month.
INTERVAL DAY [(precision)] TO SECOND	Used to represent a period of time as days, hours, minutes, and seconds. The precision specifies the precision needed for the day field, and its default is 6. The precision can have values from 0 to 9. Larger precision allows the difference between the dates to be larger. This datatype can be used to store the difference between two date time values, with seconds precision.
RAW (<size>)	Variable-length datatype used to store unstructured data, without a character set conversion. Provided for backward compatibility. Use BLOB or BFILE instead.
LONG RAW	Same as RAW, can store up to 2GB of binary data. LONG RAW is supported since Oracle9i for backward compatibility; you must use BLOB instead.
BLOB	Stores up to 4GB of unstructured binary data.
CLOB	Stores up to 4GB of character data.
NCLOB	Stores up to 4GB of Unicode character data.
BFILE	Stores unstructured binary data in operating system files outside the database. The external file size can be up to 4GB. Oracle stores only the file pointer in the database; the actual file is in the operating system.
ROWID	Stores binary data representing a physical row address of a table's row. Occupies 10 bytes.

UROWID	Stores binary data representing any type of row address: physical, logical, or foreign. Up to 4000 bytes.

Collection types are used to represent more than one element, such as an array. There are two collection datatypes: VARRAY and TABLE. Elements in the VARRAY datatype are ordered and have a maximum limit. Elements in a TABLE datatype (nested table) are not ordered, and there is no upper limit to the number of elements, unless restricted by available resources.

REF is the relationship datatype, which defines a relationship with other objects by using a reference. It actually stores pointers to data stored in different object tables.

Specifying Storage

Objective: Manage storage structures within a table.

If you create a table without specifying the storage parameters and tablespace, the table will be created in the default tablespace of the user, and the storage parameters used will be those of the default specified for the tablespace. It is always better to estimate the size of the table and specify appropriate storage parameters when creating the table. If the table is too large, you might need to consider partitioning (discussed later) or creating the table in a separate tablespace to help manage the table.

Oracle allocates a segment to the table when the table is created. This segment will have the number of extents specified by the storage parameter MINEXTENTS. Oracle allocates new extents to the table as required. Although you can have an unlimited number of extents for a segment, a little planning can improve the performance of the table. The presence of numerous extents affects the operations on the table, such as truncating a table or scanning a full table. A larger number of extents may cause additional I/Os in the data file and therefore may affect performance.

To create the ORDERS table using explicit storage parameters in the USER_DATA tablespace, use the following:

```
CREATE TABLE JAKE.ORDERS (
ORDER_NUM NUMBER,
ORDER_DATE DATE,
PRODUCT_CD VARCHAR2 (10),
QUANTITY NUMBER (10,3),
STATUS CHAR)
TABLESPACE USER_DATA
PCTFREE 5
PCTUSED 75
```

```
INITRANS 1
MAXTRANS 255
STORAGE (INITIAL 512K NEXT 512K PCTINCREASE 0
        MINEXTENTS 1 MAXEXTENTS 100
        FREELISTS 1 FREELIST GROUPS 1
        BUFFER_POOL KEEP);
```

The table will be owned by JAKE and will be created in the USER_DATA tablespace (JAKE should have appropriate space quota privileges in the tablespace). None of the storage parameters are mandatory to create a table; Oracle assigns default values if you omit them. Let's discuss the clauses used in the table creation.

The TABLESPACE clause specifies where the table is to be created. If you omit the STORAGE clause or any parameters in the STORAGE clause, the default is taken from the tablespace's default storage (if applicable). If you omit the TABLESPACE clause, the table is created in the default tablespace of the user.

The PCTFREE and PCTUSED clauses are block storage parameters. The PCTFREE clause specifies the amount of free space that should be reserved in each block of the table for future updates. In this example, you specify a low PCTFREE for the ORDERS table, because not many updates to the table increase the row length. PCTUSED specifies when the block should be considered for inserting new rows once the PCTFREE threshold is reached. Here we specified 75, so when the used space falls below 75 (as the result of updates or deletes), new rows will be added to the block.

The INITRANS and MAXTRANS clauses specify the number of concurrent transactions that can update each block of the table. Oracle reserves space in the block header for the INITRANS number of concurrent transactions. For each additional concurrent transaction, Oracle allocates space from the free space — which has an overhead of dynamically allocating transaction entry space. If the block is full, and no space is available, the transaction waits until a transaction entry space is available. MAXTRANS specifies the maximum number of concurrent transactions that can touch a block. This specification prevents unnecessarily allocating transaction space in the block header, because the transaction space allocated is never reclaimed. In most cases, the Oracle defaults of INITRANS 1 and MAXTRANS 255 are sufficient.

The STORAGE clause specifies the extent sizes, free lists, and buffer pool values. In a previous lesson we discussed the INITIAL, NEXT, MINEXTENTS, MAXEXTENTS, and PCTINCREASE parameters. These five parameters control the size of the extents allocated to the table. If the table is created on a locally managed uniform extent tablespace, these storage parameters are ignored.

The FREELIST GROUPS clause specifies the number of free list groups that should be created for the table. The default and minimum value is 1. Each free list group uses one data block (that's why the minimum value for INITIAL is two database blocks) known as the

segment header, which contains information about the extents, free blocks, and high-water mark of the table.

The FREELISTS clause specifies the number of lists for each free list group. The default and minimum value is 1. The free list manages the list of blocks that are available to add new rows. A block is removed from the free list if the free space in the block is below PCTFREE. The block remains out of the free list as long as the used space is above PCTUSED. Create more free lists if the volume of inserts to the table is high. An appropriate number would be the number of concurrent transactions performing inserts to the table.

Oracle recommends having FREELISTS and INITRANS be the same value.

The FREELIST GROUPS parameter is mostly used for parallel server configuration, in which you can specify a group for each instance.

The BUFFER_POOL parameter of the STORAGE clause specifies the area of the database buffer cache to keep the blocks of the table when read from the data file while querying or for update/delete. There are three buffer pools: KEEP, RECYCLE, and DEFAULT. The default value is DEFAULT. Specify KEEP if the table is small and is frequently accessed. The blocks in the KEEP pool are always available in the data buffer cache of SGA (System Global Area), so I/O will be faster. The blocks assigned to the RECYCLE buffer pool are removed from memory as soon as they are not needed. Specify RECYCLE for large tables or tables that are seldom accessed. If you do not specify KEEP or RECYCLE, the blocks are assigned to the DEFAULT pool, where they will be aged out using an LRU algorithm.

If you create the tablespace with the SEGMENT SPACE MANAGEMENT AUTO clause, the parameters PCTUSED, FREELISTS, and FREELIST GROUPS are ignored.

Storing LOB Structures

A table can contain columns of type CLOB, BLOB, or NCLOB. These internal large object (LOB) columns can have storage settings that are different from those of the table, and these settings can be stored in a different tablespace for easy management and performance improvement. The following example specifies storage for a LOB column when creating the table.

```
CREATE TABLE LICENSE_INFO
(DRIVER_ID VARCHAR2 (20),
DRIVER_NAME VARCHAR2 (30),
DOB DATE,
PHOTO BLOB)
TABLESPACE APP_DATA STORAGE (INITIAL 4M NEXT 4M PCTINCREASE 0)
LOB (PHOTO) STORE AS PHOTO_LOB
    (TABLESPACE APP_LARGE_DATA
    DISABLE STORAGE IN ROW
```

```
STORAGE (INITIAL 128M NEXT 128M PCTINCREASE 0)
CHUNK 4000
PCTVERSION 20
NOCACHE LOGGING);
```

The table LICENSE_INFO is created with a BLOB datatype column. The table is stored in the APP_DATA tablespace, and the BLOB column PHOTO is stored in the APP_LARGE_DATA tablespace. Let's look at the various clauses specified for the LOB storage.

The lob segment is given the name of PHOTO_LOB. If a name is not given, Oracle generates a name. You can specify multiple LOB columns in parentheses following the LOB keyword, if they all have the same storage characteristics. In such cases, you cannot specify a name for the LOB segment.

For example, if the table has three LOB columns and all have the same characteristics, you may specify the following:

```
LOB (PHOTO, VIDEO, AUDIO) STORE AS
(TABLESPACE APP_LARGE_DATA
CACHE READS NOLOGGING);
```

The TABLESPACE clause specifies the tablespace where the lob segment(s) should be stored. The tablespace can be managed locally or by the dictionary. If the LOB column is larger than 4000 bytes, data is stored in the LOB segment. Storing data in the LOB segment is known as out of line storage. If the LOB column data is less than 4000 bytes, it is stored inline, along with the other column data of the table. If you omit the TABLESPACE clause, the LOB segment is created in the table's tablespace.

The DISABLE/ENABLE STORAGE IN ROW clause specifies whether LOB data should be stored inline or out of line. ENABLE is the default and stores LOB data along with the other columns if the LOB data is less than 4000 bytes.

DISABLE stores the LOB data in the LOB segment regardless of it size. Whether the LOB data is stored inline or out of line, the LOB locator is always stored along with the row.

The STORAGE clause specifies the extent sizes and growth parameters. These parameters are the same, as those you would use with a table.

The CHUNK clause specifies the total number of bytes of data that will be read or written during LOB manipulation. CHUNK must be a multiple of the database block size. If you specify a value other than a multiple of the block size, Oracle uses the next higher value that is a multiple of block size. For example, if you specify 4000 for CHUNK and the database block size is 2048, Oracle will take the value of 4096. The default value for CHUNK is the database block size, and the maximum value is 32KB. The INITIAL and NEXT values specified in the STORAGE clause must be higher than the value for CHUNK.

The PCTVERSION clause specifies the percentage of all used LOB data space that can be occupied by old versions of LOB data pages. Since LOB data changes are not written to the rollback segments, PCTVERSION specifies the percentage of old information that should be kept in the LOB segment for consistent reads. The default is 10, and the percentage can range from 0 through 100.

The CACHE / NOCACHE / CACHE READS clause specifies whether to cache the LOB reads. If the LOB is read and updated frequently, use the CACHE clause. NOCACHE is the default, and it is useful for a LOB that is read infrequently and never updated. CACHE READS caches only the read operation, which is useful for a LOB that is read frequently, but never updated.

The LOGGING / NOLOGGING clause specifies whether redo information should be generated for LOB data. NOLOGGING does not write redo and is useful for faster data loads. You cannot specify CACHE and NOLOGGING together.

Creating a Table from a Query

Objective: Create regular and temporary tables.

You can create a table using existing tables or views by specifying a subquery instead of defining the columns. The subquery can refer to more than one table or view. The table is created with the rows returned from the subquery. You can specify new column names for the table, but Oracle derives the datatype and maximum width based on the query result — you cannot specify the datatype with this method. You can specify the storage parameters for the tables created by using the subquery. For example, let's create a new table from the ORDERS table for the orders that are accepted. Notice that the new column names are specified.

```
CREATE TABLE ACCEPTED_ORDERS
        (ORD_NUMBER, ORD_DATE, PRODUCT_CD, QTY)
        TABLESPACE USERS
        PCTFREE 0
        STORAGE (INITIAL 128K NEXT 128K PCTINCREASE 0)
AS
        SELECT ORDER_NUM, ORDER_DATE, PRODUCT_CD, QUANTITY
        FROM ORDERS
        WHERE STATUS = 'A';
```

The CREATE TABLE...AS SELECT... will not work if the query refers to columns of LONG datatype. When you create a table using the subquery, only the NOT NULL constraints

associated with the columns are copied to the new table. Other constraints and column default definitions are not copied.

Partitioning Tables

When tables are very large, you can manage them better by using *partitioning*. Partitioning is breaking a large table into manageable pieces based on the values in a column (or multiple columns) known as the partition key.

If you have a very large table spread across many data files, and one disk fails, you have to recover the entire table. However, if the table is partitioned, you need to recover only that partition. SQL statements can access the required partition(s) rather than reading the entire table. Four partitioning methods are available:

Range You can create a range partition in which the partition key values are in a range — for example, you can partitions a transaction table on the transaction date, and you can create a partition for each month or each quarter. The partition column list can be one or more columns.

Hash Hash partitions are more appropriate when you do not know how much data will be in a range or whether the sizes of the partition vary. Hash partitions use a hash algorithm on the partitioning columns. The number of partitions required should be specified preferably as a power of two (such as 2, 4, 8, 16, and so on).

List If you know all the values that are supposed to be stored in the column and want to create a partition for each value, use the list partition method. You must specify a list of values when defining the partition, and you can group one or more values together. List partitioning gives explicit control over how each row maps to a partition, whereas in range partition a range of values map to a partition. List partitioning is good for managing partitions using discrete values, which may not be possible using range partitioning.

Composite This method uses the range partition method to create partitions and the hash partition method to create subpartitions.

The *logical attributes* for all partitions remain the same (such as column name, datatype, constraints, and so on), but each partition can have its own *physical attributes* (such as tablespace, storage parameters, and so on). Each partition in the partitioned table is allocated a segment. You can place these partitions on different tablespaces, which can help you to balance the I/O by placing the data files appropriately on disk. Also, by having the partitions in different tablespaces, you can make a partition tablespace read-only. You can specify the storage parameters at the table level or for each partition.

Warning: Partitioned tables cannot have any columns with LONG or LONG RAW datatypes.

Range-Partitioned Table

To create a range-partitioned table, you specify the PARTITION BY RANGE clause in the CREATE TABLE command. As stated earlier, range partitioning is suitable for tables that have column(s) with a range of values. For example, your transaction table might have the transaction date column, with which you can create a partition for every month or for a quarter. Consider the following example:

```
CREATE TABLE ORDER_TRANSACTION (
  ORD_NUMBER NUMBER(12),
  ORD_DATE DATE,
  PROD_ID VARCHAR2 (15),
  QUANTITY NUMBER (15,3))
PARTITION BY RANGE (ORD_DATE)
(PARTITION FY2001Q4 VALUES LESS THAN
      (TO_DATE('01012002','MMDDYYYY'))
      TABLESPACE ORD_2001Q4,
PARTITION FY2002Q1 VALUES LESS THAN
      (TO_DATE('04012002','MMDDYYYY'))
      TABLESPACE ORD_2002Q1 STORAGE (INITIAL 500M NEXT 500M)
      INITRANS 2 PCTFREE 0,
PARTITION FY2002Q2 VALUES LESS THAN
      (TO_DATE('07012002','MMDDYYYY'))
      TABLESPACE ORD_2002Q2,
PARTITION FY2000Q3 VALUES LESS THAN
      (TO_DATE('10012002','MMDDYYYY'))
      TABLESPACE ORD_2002Q3 STORAGE (INITIAL 10M NEXT 10M))
STORAGE (INITIAL 200M NEXT 200M PCTINCREASE 0 MAXEXTENTS 4096)
NOLOGGING;
```

This example creates a range-partitioned table named ORDER_TRANSACTION. PARTITION BY RANGE specifies that the table be range partitioned; the partition column is provided in parentheses (separate multiple columns with commas), and the partition specifications are defined. Each partition specification begins with the keyword PARTITION. You can optionally provide a name for the partition. The VALUES LESS THAN clause defines values for the partition columns that should be in the partition. In the example, each

partition is created on different tablespaces; partitions FY2001Q4 and FY2002Q2 inherit the storage parameter values from the table definition, whereas FY2002Q1 and FY2002Q3 have the storage parameters explicitly defined.

Records with ORD_DATE prior to 01-Jan-2002 will be stored in partition FY2001Q4; since you did not specify a partition for records with ORD_DATE after 31-Sep-2002, Oracle rejects those rows. NULL value is treated greater than all other values. If the partition key can have NULL values or records with a higher ORD_DATE than the highest upper range in the partition specification list, you must create a partition for the upper range. The MAXVALUE parameter specifies that the partition bound is infinite. In this example, an upper-bound partition can be specified as follows:

```
CREATE TABLE ORDER_TRANSACTION ( )
PARTITION BY RANGE (ORD_DATE)
(PARTITION FY1999Q4 VALUES LESS THAN
      (TO_DATE('01012002','MMDDYYYY'))
      TABLESPACE ORD_2001Q4,
PARTITION FY2999Q4 VALUES LESS THAN (MAXVALUE)
      TABLESPACE ORD_2999Q4 )
STORAGE (INITIAL 200M NEXT 200M PCTINCREASE 0 MAXEXTENTS 4096)
NOLOGGING;
```

Hash-Partitioned Table

You create a hash-partitioned table by specifying the PARTITION BY HASH clause in the CREATE TABLE command. Hash partitioning is suitable for any large table to take advantage of Oracle9i's performance improvements even if you do not have column(s) with a range of values. Hash partitioning is suitable when you do not know how many rows will be in the table. Choose a column with unique values or more distinct values for the partition key.

The following example creates a hash-partitioned table with four partitions. The partitions are created in tablespaces DOC101, DOC102, and DOC103. Since there are four partitions and only three tablespaces listed, Oracle reuses the DOC101 tablespace for the fourth partition. Oracle creates the partition names as SYS_XXXX. Physical attributes are specified at the table level only.

```
CREATE TABLE DOCUMENTS1 (
  DOC_NUMBER NUMBER(12),
  DOC_TYPE VARCHAR2 (20),
  CONTENTS VARCHAR2 (600))
PARTITION BY HASH (DOC_NUMBER, DOC_TYPE)
```

```
PARTITIONS 4 STORE IN (DOC101, DOC102, DOC103)
STORAGE (INITIAL 64K NEXT 64K PCTINCREASE 0 MAXEXTENTS 4096);
```

The following example creates a hash-partitioned table with named partitions in the tablespaces.

```
CREATE TABLE DOCUMENTS2 (
  DOC_NUMBER NUMBER(12),
  DOC_TYPE VARCHAR2 (20),
  CONTENTS VARCHAR2 (600))
PARTITION BY HASH (DOC_NUMBER, DOC_TYPE)
( PARTITION DOC201 TABLESPACE DOC201,
 PARTITION DOC202 TABLESPACE DOC202,
 PARTITION DOC203 TABLESPACE DOC203,
 PARTITION DOC203 TABLESPACE DOC204 )
STORAGE (INITIAL 64K NEXT 64K PCTINCREASE 0 MAXEXTENTS 4096);
```

List-Partitioned Table

To create a list-partitioned table, you specify the PARTITION BY LIST clause in the CREATE TABLE command. List partitioning gives explicit control over the rows that are stored in each partition. You can specify NULL as a valid list value. If you insert a row that has a partition column value not defined in the list, Oracle rejects the row. You can specify only one column name as the partition key. The following example creates a list-partitioned table.

```
CREATE TABLE POPULATION_STATS
(STATE VARCHAR2 (2),
COUNTY VARCHAR2 (30),
CITY VARCHAR2 (30),
MEN NUMBER,
WOMEN NUMBER,
BCHILD NUMBER,
GCHILD NUMBER)
PARTITION BY LIST (STATE)
(PARTITION SC VALUES ('TX','LA','OK') TABLESPACE SC_DATA,
```

```
PARTITION SW VALUES ('NM','AZ') TABLESPACE SW_DATA,
PARTITION SE VALUES ('AR','MS','AL') TABLESPACE SE_DATA);
```

Composite-Partitioned Table

Composite partitions have range partitions and hash subpartitions. Only subpartitions are physically created on the disk (tablespace); partitions are logical representations only. Composite partitioning gives the flexibility of range and hash for tables with a smaller range of values.

In the following example, the table is range partitioned on the MAKE_YEAR column; each partition is subdivided based on the MODEL into 4 subpartitions on 4 tablespaces. Each tablespace will have one subpartition from each partition, that is, 3 subpartitions per tablespace, thereby having 12 subpartitions.

```
CREATE TABLE CARS (
MAKE_YEAR NUMBER(4),
MODEL VARCHAR2 (30),
MANUFACTR VARCHAR2 (50),
QUANTITY NUMBER)
PARTITION BY RANGE (MAKE_YEAR)
SUBPARTITION BY HASH (MODEL) SUBPARTITIONS 4
STORE IN(TSMK1, TSMK2, TSMK3, TSMK4)
( PARTITION M2001 VALUES LESS THAN (2002),
PARTITION M2002 VALUES LESS THAN (2003),
PARTITION M2001 VALUES LESS THAN (MAXVALUE))
STORAGE (INITIAL 64K NEXT 64K PCTINCREASE 0 MAXEXTENTS 4096);
```

The following example shows how to name the subpartitions and store each subpartition in a different tablespace. Subpartitions for partition M2001 have explicit storage parameters specified.

```
CREATE TABLE CARS2 (
MAKE_YEAR NUMBER(4),
MODEL VARCHAR2 (30),
MANUFACTR VARCHAR2 (50),
QUANTITY NUMBER)
PARTITION BY RANGE (MAKE_YEAR)
```

```
SUBPARTITION BY HASH (MODEL) SUBPARTITIONS 4
( PARTITION M2001 VALUES LESS THAN (2002)
STORAGE (INITIAL 128K NEXT 128K)
( SUBPARTITION M2001_SP1 TABLESPACE TS011,
SUBPARTITION M2001_SP2 TABLESPACE TS012,
SUBPARTITION M2001_SP3 TABLESPACE TS013,
SUBPARTITION M2001_SP4 TABLESPACE TS014 ),
PARTITION M2002 VALUES LESS THAN (2003)
( SUBPARTITION M2002_SP1 TABLESPACE TS021,
SUBPARTITION M2002_SP2 TABLESPACE TS022,
SUBPARTITION M2002_SP3 TABLESPACE TS023,
SUBPARTITION M2002_SP4 TABLESPACE TS024 ),
PARTITION M9999 VALUES LESS THAN (MAXVALUE)
( SUBPARTITION M2999_SP1 TABLESPACE TS991,
SUBPARTITION M2999_SP2 TABLESPACE TS992,
SUBPARTITION M2999_SP3 TABLESPACE TS993,
SUBPARTITION M2999_SP4 TABLESPACE TS994 ))
STORAGE (INITIAL 64K NEXT 64K PCTINCREASE 0 MAXEXTENTS 4096);
```

Using Other Create Clauses

You can specify the following additional clauses while creating a table. These clauses help to manage various types of operations on the table.

LOGGING/NOLOGGING LOGGING is the default for the table and tablespace, but if the tablespace is defined as NOLOGGING, the table uses NOLOGGING. LOGGING specifies that table creation and direct-load inserts should be logged to the redo log files. Creating the table by using a subquery and the NOLOGGING clause can dramatically reduce the time to create large tables. If the table creation, initial data population (using a subquery), and direct-load inserts are not logged to the redo log files when using the NOLOGGING clause, you must back up the table (or better yet, the entire tablespace) after such operations are performed. Media recovery will not create or load tables created with the NOLOGGING attribute. You can also specify a separate LOGGING or NOLOGGING attribute for indexes and LOB storage of the table, independent of the table's attribute. The following example creates a table with the NOLOGGING clause.

```
CREATE TABLE MY_ORDERS (... ...)
```

```
TABLESPACE USER_DATA STORAGE (... ...)
NOLOGGING;
```

PARALLEL/NOPARALLEL NOPARALLEL is the default. PARALLEL causes the table creation (if created using a subquery) and the DML statements on the table to execute in parallel. Normally, a single-server process performs operations on tables in a transaction (serial operation). When the PARALLEL attribute is set, Oracle uses multiple processes to complete the operation for a full table scan. You can specify a degree for the parallelism; if a degree is not specified, Oracle calculates the optimum degree of parallelism. The parameter PARALLEL_THREADS_PER_CPU determines the number for parallel degree per CPU; usually the default is 2. If you do not specify the degree, Oracle calculates the degree based on this parameter and the number of CPUs available. The following example creates a table by using a subquery. The table creation will not be logged in the redo log file, and multiple processes will query the JAKE.ORDERS table and create the MY_ORDERS table.

```
CREATE TABLE MY_ORDERS (... ...)
TABLESPACE USER_DATA STORAGE (... ...)
NOLOGGING PARALLEL
AS SELECT * FROM JAKE.ORDERS;
```

CACHE/NOCACHE NOCACHE is the default. For small look-up tables that are frequently accessed, you can specify the CACHE clause to have the blocks retrieved using a full table scan placed at the MRU end of the LRU list in the buffer cache; the blocks are not aged out of the buffer cache immediately. The default behavior (NOCACHE) is to place the blocks from a full table scan at the tail end of the LRU list, where they are moved out of the list as soon as a different process or query needs these blocks for storing another table's blocks in the cache.

Creating Temporary Tables

Temporary tables hold information that is available only to the session that created the data. The definition of the temporary table is available to all sessions.

To create a temporary table, you use the CREATE GLOBAL TEMPORARY TABLE statement. The data in the table can be session-specific or transaction-specific. The ON COMMIT clause specifies which. The following statement creates a temporary table that is transaction-specific.

```
CREATE GLOBAL TEMPORARY TABLE INVALID_ORDERS
(ORDER# NUMBER (8),
ORDER_DT DATE,
VALUE NUMBER (12,2))
ON COMMIT DELETE ROWS;
```

Oracle deletes rows or truncates the table after each commit. To define the table as session-specific, use the ON COMMIT PRESERVE ROWS clause.

Storage for temporary tablespace is allocated in the temporary tablespace of the user. Segments are created only when the first insert statement is performed on the table. The temporary segments allocated to temporary tables are deallocated at the end of the transaction for transaction-specific tables and at the end of session for session-specific tables.

You can create indexes on temporary tables. DML (Data Manipulation Language) statements on temporary tables do not generate redo information, but undo information is generated.

1.2. Altering Tables

Objective: Reorganize, truncate, and drop a table.

You can alter a table by using the ALTER TABLE command to change it's the table's storage settings, add or drop columns, or modify the column characteristics such as default value, datatype, and length. You can also move the table from one tablespace to another, disable constraints and triggers, and change the clauses such as PARALLEL, CACHE, and LOGGING. In this section, we will discuss altering the storage and space used by the table.

You cannot change the STORAGE parameters INITIAL and MINEXTENTS by using the ALTER TABLE command. You can change NEXT, PCTINCREASE, MAXEXTENTS, FREELISTS, and FREELIST GROUPS. The changes will not affect the extents that are already allocated. When you change NEXT, the next extent allocated will have the new size. When you alter PCTINCREASE, the next extent allocated will be based on the current value of NEXT, but further extent sizes will be calculated with the new PCTINCREASE value. Here is an example of changing the storage parameters:

```
ALTER TABLE ORDERS
STORAGE (NEXT 512K PCTINCREASE 0 MAXEXTENTS UNLIMITED);
```

Allocating and Deallocating Extents

You can allocate new extents to a table or a partition manually by using the ALTER TABLE command. You can optionally specify a filename if you want to allocate the extent in a particular data file. You can specify the size of the extent in bytes (use K or M to specify the size in KB or MB). If you omit the size of the extent, Oracle uses the NEXT size. For example, to manually allocate the next extent, use the following:

```
ALTER TABLE ORDERS ALLOCATE EXTENT;
```

To specify the size of the extent to be allocated, use the following:

```
ALTER TABLE ORDERS ALLOCATE EXTENT SIZE 200K;
```

To specify the data file where the extent should be allocated, use the following:

```
ALTER TABLE ORDERS ALLOCATE EXTENT SIZE 200K
DATAFILE 'C:\ORACLE\ORADATA\USER_DATA01.DBF';
```

The data file should belong to the tablespace where the table or the partition resides.

Sometimes the storage space estimated for the table is too high. The table may be created with large extent sizes, or if you do not set the PCTINCREASE size properly, the space allocated to the table may be large. Any other table or object cannot use the space once allocated. You can free up such unused free space by manually deallocating the unused blocks above the high-water mark (HWM) of the table. The HWM indicates the historically highest amount of used space in a segment. The HWM moves only in the forward direction, that is, when new blocks are used to store data in a table, the HWM is increased. When rows are deleted, and even if a block is completely empty, the HWM is not decreased. The HWM is reset only when you TRUNCATE the table.

You can use the UNUSED_SPACE procedure of the DBMS_SPACE package to find the HWM of the segment. The following listing shows the parameters for the procedure.

```
PROCEDURE       UNUSED_SPACE
Argument Name                  Type                 In/Out    Default?
------------------------------ ---------            -------   --------
SEGMENT_OWNER                  VARCHAR2             IN
SEGMENT_NAME                   VARCHAR2             IN
SEGMENT_TYPE                   VARCHAR2             IN
TOTAL_BLOCKS                   NUMBER               OUT
TOTAL_BYTES                    NUMBER               OUT
UNUSED_BLOCKS                  NUMBER               OUT
UNUSED_BYTES                   NUMBER               OUT
LAST_USED_EXTENT_FILE_ID       NUMBER               OUT
LAST_USED_EXTENT_BLOCK_ID      NUMBER               OUT
LAST_USED_BLOCK                NUMBER               OUT
PARTITION_NAME                 VARCHAR2             IN        DEFAULT
```

You can execute the procedure by specifying the owner, type (table, index, table partition, table subpartition, and so on), and name. The HWM is TOTAL_BYTES – UNUSED_BYTES. Here is an example:

```
SQL> variable vtotalblocks number
```

```
SQL> variable vtotalbytes number
SQL> variable vunusedblocks number
SQL> variable vunusedbytes number
SQL> variable vlastusedefid number
SQL> variable vlastusedebid number
SQL> variable vlastusedblock number
SQL> EXECUTE DBMS_SPACE.UNUSED_SPACE ('JOHN', -
> 'ORDERS', 'TABLE', :vtotalblocks, :vtotalbytes, -
> :vunusedblocks, :vunusedbytes, -
> :vlastusedefid, :vlastusedebid, :vlastusedblock);

PL/SQL procedure successfully completed.

SQL> select :vtotalbytes, :vunusedbytes from dual;

:VTOTALBYTES     :VUNUSEDBYTES
----------------     -------------
       1224288      8507904
SQL>
```

You can use the DEALLOCATE UNUSED clause of the ALTER TABLE command to free up the unused space allocated to the table. For example, to free up all blocks above the HWM, use this statement:

```
ALTER TABLE ORDERS DEALLOCATE UNUSED;
```

You can use the KEEP parameter in the UNUSED clause to specify the number of blocks you want to keep above the HWM after deallocation. For example, to have 100KB of free space available for the table above the HWM, specify the following:

```
ALTER TABLE ORDERS DEALLOCATE UNUSED KEEP 100K;
```

If you do not specify KEEP, and the HWM is below MINEXTENTS, Oracle keeps MINEXTENTS extents. If you specify KEEP, and the HWM is below MINEXTENTS,

Oracle adjusts the MINEXTENTS to match the number of extents. If the HWM is less than the size of INITIAL, and KEEP is specified, Oracle adjusts the size of INITIAL.

Table 2 shows some examples of freeing up space. Let's assume that the table is created with (INITIAL 1024K NEXT 1024K PCTINCREASE 0 MINEXTENTS 4) and now the table has 10 extents (total size 10,240KB).

Table 2. DEALLOCATE Clause Examples.

HWM	DEALLOCATE Clause	Resulting Size	Extent Count
7000KB	UNUSED;	7000KB	Seven; the seventh extent will be split at the HWM.
200KB	UNUSED;	4096KB	Four, because the KEEP clause is not specified.
200KB	UNUSED KEEP 100K;	300KB	One; the initial extent is split at the HWM.
2000KB	UNUSED KEEP 0K;	2000KB	Two; the second extent is split at the HWM.

TIP: When a full table scan is performed, Oracle reads each block up to the table's high-water mark.

You use the TRUNCATE command to delete all rows of the table and to reset the HWM of the table. You can keep the space allocated to the table or deallocate the extents when using TRUNCATE. By default, Oracle deallocates all the extents allocated above MINEXTENTS of the table and the associated indexes. To preserve the space allocated, you must specify the REUSE clause (DROP is the default), as in this example:

```
TRUNCATE TABLE ORDERS REUSE STORAGE;
```

Warning: To truncate a table, you must disable all referential integrity constraints.

Reorganizing Tables

You can use the MOVE clause of the ALTER TABLE command on a nonpartitioned table to reorganize or to move from one tablespace to another. You can reorganize the table to reduce the number of extents by specifying larger extent sizes or to prevent row migration. When you

move a table, Oracle creates a new segment for the table, copies the data, and drops the old segment. The new segment can be in the same tablespace or in a different tablespace.

Since the old segment is dropped only after you create the new segment, you need to make sure you have sufficient space in the tablespace, if you're not changing to a different tablespace. The MOVE clause can specify a new tablespace, new storage parameters for the table, new free space management parameters, and new transaction entry parameters. You can use the NOLOGGING clause to speed up the reorganization by not writing the changes to the redo log file.

The following example moves the ORDERS table to another tablespace named NEW_DATA. New storage parameters are specified, and the operation is not logged in the redo log files (NOLOGGING).

```
ALTER TABLE ORDERS MOVE
TABLESPACE NEW_DATA
STORAGE (INITIAL 50M NEXT 5M PCTINCREASE 0)
PCTFREE 0 PCTUSED 50
INITRANS 2 NOLOGGING;
```

Prior to Oracle8i, you reorganized a table using the export-drop-import method. Queries are allowed on the table while the move operation is in progress, but no insert, update, or delete operations are allowed. The granted permissions on the table are retained.

Dropping a Table

If a table is no longer used, you can drop it to free up space. Once you drop a table, the action cannot be undone. The syntax follows:

```
DROP TABLE [schema.]table_name [CASCADE CONSTRAINTS]
```

When you drop a table, the data and definition of the table are removed. The indexes, constraints, triggers, and privileges on the table are also dropped. Oracle does not drop the views, materialized views, or other stored programs that reference the table, but it marks them as invalid. You must specify the CASCADE CONSTRAINTS clause if there are referential integrity constraints referring to the primary key or unique key of this table. Here's how to drop the table TEST owned by user SCOTT:

```
DROP TABLE SCOTT.TEST;
```

Truncating a Table

The TRUNCATE statement is similar to the DROP statement, but it does not remove the structure of the table, so none of the indexes, constraints, triggers, and privileges on the table are dropped. By default, the space allocated to the table and indexes is freed. If you do not

want to free up the space, include the REUSE STORAGE clause. You cannot roll back a truncate operation. Also, you cannot selectively delete rows using the TRUNCATE statement.

The syntax of TRUNCATE statement is:

```
TRUNCATE {TABLE|CLUSTER} [<schema>.]<name>
[{DROP|REUSE} STORAGE]
```

You cannot truncate the parent table of an enabled referential integrity constraint. You must first disable the constraint and then truncate the table, even if the child table has no rows. The following example demonstrates this process:

```
SQL> CREATE TABLE t1 (
  2     t1f1 NUMBER CONSTRAINT pk_t1 PRIMARY KEY);
Table created.

SQL> CREATE TABLE t2 (t2f1 NUMBER CONSTRAINT fk_t2
                              REFERENCES t1 (t1f1));
Table created.

SQL> TRUNCATE TABLE t1;
truncate table t1
               *
ERROR at line 1:
ORA-02266: unique/primary keys in table referenced by enabled foreign keys

SQL> ALTER TABLE t2 DISABLE CONSTRAINT fk_t2;
Table altered.

SQL> TRUNCATE TABLE t1;
Table truncated.

SQL>
```

TRUNCATE versus DELETE

The TRUNCATE statement is similar to a DELETE statement without a WHERE clause, except for the following:

- TRUNCATE is very fast on both large and small tables. DELETE will generate undo information, in case a rollback is issued, but TRUNCATE will not generate undo.
- TRUNCATE is DDL and, like all DDL, performs an implicit commit — you cannot roll back a TRUNCATE. Any uncommitted DML changes will also be committed with the TRUNCATE.
- TRUNCATE resets the high-water mark in the table and all indexes. Since full table scans and index fast-full scans read all data blocks up to the high-water mark, full-scan performance after a DELETE will not improve; after a TRUNCATE, performance will be very fast.
- TRUNCATE does not fire any DELETE triggers.
- There is no object privilege that can be granted to allow a user to truncate another user's table. The DROP ANY TABLE system privilege is required to truncate a table in another schema.
- When a table is truncated, the storage for the table and all indexes can be reset back to the initial size. A DELETE will never shrink the size of a table or its indexes.

TRUNCATE versus DROP TABLE

Using TRUNCATE is also different from dropping and re-creating a table. Compared with dropping and recreating a table, TRUNCATE does not do the following:

- Invalidate dependent objects
- Drop indexes, triggers, or referential integrity constraints
- Require privileges to be re-granted

TIP: Use the TRUNCATE statement to delete all rows from a large table; it does not write the rollback entries and is much faster than the DELETE statement when deleting a large number of rows.

Dropping Columns

Objective: Drop a column within a table.

You can drop a column that is not used immediately, or you can mark the column as not used and drop it later.

Here is the syntax for dropping a column:

```
ALTER TABLE [<schema>.]<table_name>
```

```
DROP {COLUMN <column_name> | (<column_names>)}
[CASCADE CONSTRAINTS]
```

DROP COLUMN drops the column name specified from the table. You can provide more than one column name separated by commas inside parentheses.

The indexes and constraints on the column are also dropped. You must specify CASCADE CONSTRAINTS if the dropped column is part of a multicolumn constraint; the constraint will be dropped.

Here is the syntax for marking a column as unused:

```
ALTER TABLE [<schema>.]<table_name>
SET UNUSED {COLUMN <column_name> | (<column_names>)}
[CASCADE CONSTRAINTS]
```

You usually mark a column as unused instead of dropping it immediately if the table is very large and consumes a lot of resources at peak hours. In such cases, you would mark the column as unused and drop it at off-peak hours. Once the column is marked as unused, you will not see it as part of the table definition. Let's mark the UPDATE_DT column in the ORDERS table as unused:

```
ALTER TABLE orders SET UNUSED COLUMN update_dt;
```

The syntax for dropping a column already marked as unused is:

```
ALTER TABLE [<schema>.]<table_name>
DROP {UNUSED COLUMNS | COLUMNS CONTINUE}
```

Use the COLUMNS CONTINUE clause to continue a DROP operation that was previously interrupted. You cannot specify selected column names to drop after marking the column as unused. The DROP UNUSED COLUMNS clause will drop all the columns that are marked as unused. To clear data from the UPDATE_DT column from the ORDERS table, use the following statement:

```
ALTER TABLE orders DROP UNUSED COLUMNS;
```

1.3. Analyzing Tables

You can analyze a table to verify the blocks in it, to find the chained and migrated rows, and to collect statistics on the table. You can specify the PARTITION or SUBPARTITION clause to analyze a specific partition or subpartition of the table.

Validating Structure

As the result of hardware problems, disk errors, or software bugs, some blocks can become corrupted (logical corruption). Oracle returns a corruption error only when the rows are accessed. (The Export utility identifies logical corruption in tables, because it does a full table scan.) You can use the ANALYZE command to validate the structure or check the integrity of the blocks allocated to the table. If Oracle finds blocks or rows that are not readable, it returns an error. The ROWIDs of the bad rows are inserted into a table.

You can specify the name of the table in which you want the ROWIDs to be saved; by default, Oracle looks for the table named INVALID_ROWS. You can create the table using the script utlvalid.sql supplied from Oracle, located in the rdbms/admin directory of the software installation. The structure of the table is as follows:

```
SQL> @c:\oracle\ora90\rdbms\admin\utlvalid.sql

Table created.

SQL> desc invalid_rows
Name                           Null?    Type
------------------------------ -------- ------------
OWNER_NAME                              VARCHAR2(30)
TABLE_NAME                              VARCHAR2(30)
PARTITION_NAME                          VARCHAR2(30)
SUBPARTITION_NAME                       VARCHAR2(30)
HEAD_ROWID                              ROWID
ANALYZE_TIMESTAMP                       DATE
```

This example validates the structure of the ORDERS table:

```
ANALYZE TABLE ORDERS VALIDATE STRUCTURE;
```

If Oracle encounters bad rows, it inserts them into the INVALID_ROWS table. To specify a different table name, use the following syntax.

```
ANALYZE TABLE ORDERS VALIDATE STRUCTURE
INTO SCOTT.CORRUPTED_ROWS;
```

You can also validate the blocks of the indexes associated with the table by specifying the CASCADE clause.

```
ANALYZE TABLE ORDERS VALIDATE STRUCTURE CASCADE;
```

To analyze a partition by name MAY2002 in table GLEDGER, specify

```
ANALYZE TABLE GLEDGER PARTITION (MAY2002) VALIDATE STRUCTURE;
```

Finding Migrated Rows

A row is *migrated* if the row is moved from its original block to another block because there was not enough free space available in its original block to accommodate the row, which was expanded due to an update. Oracle keeps a pointer in the original block to indicate the new block ID of the row. When there are many migrated rows in a table, performance of the table is affected, because Oracle has to read two blocks instead of one for a given row retrieval or update. You can prevent this problem by specifying an efficient PCTFREE value.

A row is *chained* if the row is bigger than the block size of the database. Normally, the rows of a table with LOB datatypes are more likely to become chained.

You can use the LIST CHAINED ROWS clause of the ANALYZE command to find the chained and migrated rows of a table. Oracle writes the ROWID of such rows to a specified table. If no table is specified, Oracle looks for the CHAINED_ROWS table. You can create this table using the script utlchain.sql supplied from Oracle, located in the rdbms/admin directory of the software installation. The structure of the table is as follows:

```
SQL> @c:\oracle\ora90\rdbms\admin\utlchain.sql
Table created.
SQL> DESC CHAINED_ROWS
Name                          Null?    Type
----------------------------- -------- ------------
OWNER_NAME                             VARCHAR2(30)
TABLE_NAME                             VARCHAR2(30)
CLUSTER_NAME                           VARCHAR2(30)
PARTITION_NAME                         VARCHAR2(30)
SUBPARTITION_NAME                      VARCHAR2(30)
HEAD_ROWID                             ROWID
ANALYZE_TIMESTAMP                      DATE
```

Only the ROWIDs are listed in the CHAINED_ROWS table. You can use this information to save these chained rows to a different table, delete them from the source table, and insert them back from the second table. Here is one way to fix migrated rows in a table:

1. Analyze the table to find migrated rows.

   ```
   ANALYZE TABLE ORDERS LIST CHAINED ROWS;
   ```

2. Find the number of migrated rows.

```
SELECT COUNT(*)
FROM CHAINED_ROWS
WHERE OWNER_NAME = 'SCOTT'
AND TABLE_NAME = 'ORDERS';
```

3. If there are migrated rows, create a temporary table to hold the migrated rows.

```
CREATE TABLE TEMP_ORDERS AS
SELECT * FROM ORDERS
WHERE ROWID IN (SELECT HEAD_ROWID
        FROM CHAINED_ROWS
        WHERE OWNER_NAME = 'SCOTT'
        AND TABLE_NAME = 'ORDERS');
```

4. Delete the rows from the ORDERS table.

```
DELETE FROM ORDERS
WHERE ROWID IN (SELECT HEAD_ROWID
        ROM CHAINED_ROWS
        WHERE OWNER_NAME = 'SCOTT'
        AND TABLE_NAME = 'ORDERS');
```

5. Insert the rows back into the ORDERS table.

```
INSERT INTO ORDERS
SELECT * FROM TEMP_ORDERS;
```

Before deleting the rows, make sure you disable any foreign key constraints referring to the ORDERS table. You will not be able to delete the rows if there are child rows, and most important, defining the constraints with the CASCADE option deletes the child rows!.

Collecting Statistics

You can collect statistics about a table and save them in the dictionary tables by using the ANALYZE command. The cost-based optimizer also uses the statistics to generate the execution plan of SQL statements. You can calculate the exact statistics (COMPUTE) of the table or sample a few rows and estimate the statistics (ESTIMATE). By default, Oracle collects statistics for all the columns and indexes in the table. For large tables, you may want to estimate, because when you compute the statistics, Oracle reads each block of the table.

The following information is collected and saved in the dictionary when you use the ANALYZE command to collect statistics:

- The total number of rows in the table and the number of chained rows
- The total number of blocks allocated, the total number of unused blocks, and the average free space in each block
- The average row length

Here is an example of analyzing a table using the COMPUTE clause:

```
ANALYZE TABLE ORDERS COMPUTE STATISTICS;
```

When using the ESTIMATE option, you can either specify a certain number of rows or specify a certain percentage of rows in the table. If the rows specified are more than 50 percent of the table, Oracle does a COMPUTE. If you do not specify the SAMPLE clause, Oracle samples 1064 rows. To specify the number of rows, use the following:

```
ANALYZE TABLE ORDERS ESTIMATE STATISTICS
SAMPLE 200 ROWS;
```

To specify a percentage of the table to sample, use the following:

```
ANALYZE TABLE ORDERS ESTIMATE STATISTICS
SAMPLE 20 PERCENT;
```

To remove statistics collected on a table, use the DELETE STATISTICS option, as follows:

```
ANALYZE TABLE ORDERS DELETE STATISTICS;
```

NOTE: You can also collect the statistics using the DBMS_STATS package. You have the option of collecting the statistics into a non-dictionary table.

1.4. Querying Table Information

Several data dictionary views are available to provide information about the tables. We will discuss certain views and their columns that you should be familiar with before taking the test.

DBA_TABLES

You primarily use the DBA_TABLES, USER_TABLES, and ALL_TABLES views to query for information about tables (TABS is a synonym for USER_TABLES). The views contain the following information (the columns that can be used in the query are provided in parentheses):

- Identity (OWNER, TABLE_NAME)
- Storage (TABLESPACE_NAME, PCT_FREE, PCT_USED, INI_TRANS, MAX_TRANS, INITIAL_EXTENT, NEXT_EXTENT, MAX_EXTENTS, MIN_EXTENTS, PCT_INCREASE, FREELISTS, FREELIST_GROUPS, BUFFER_POOL)
- Statistics (NUM_ROWS, BLOCKS, EMPTY_BLOCKS, AVG_SPACE, CHAIN_CNT, AVG_ROW_LEN, AVG_SPACE_FREELIST_BLOCKS, NUM_FREELIST_BLOCKS, SAMPLE_SIZE, LAST_ANALYZED)
- Miscellaneous create options (LOGGING, DEGREE, CACHE, PARTITIONED, NESTED)

DBA_TAB_COLUMNS

Use the DBA_TAB_COLUMNS, USER_TAB_COLUMNS, and ALL_TAB_COLUMNS views to display information about the columns in a table. You can query the following information:

- Identity (OWNER, TABLE_NAME, COLUMN_NAME, COLUMN_ID)
- Column characteristics (DATA_TYPE, DATA_LENGTH, DATA_PRECISION, DATA_SCALE, NULLABLE, DEFAULT_LENGTH, DATA_DEFAULT)
- Statistics (AVG_COL_LEN, NUM_DISTINCT, HIGH_VALUE, LOW_VALUE, NUM_NULLS, LAST_ANALYZED, SAMPLE_SIZE)

Table 3 lists other dictionary views that show information about the tables in the database.

Table 3. Dictionary Views with Table Information.

View Name	Contents
ALL_ALL_TABLES DBA_ALL_TABLES USER_ALL_TABLES	Similar information as in DBA_TABLES; shows information about relational tables and object tables.
ALL_TAB_PARTITIONS DBA_TAB_PARTITIONS USER_TAB_PARTITIONS	Partitioning information, storage parameters, and partition-level statistics gathered.
ALL_TAB_SUBPARTITIONS DBA_TAB_SUBPARTITIONS USER_TAB_SUBPARTITIONS	Subpartition information for composite partitions in the database.
ALL_OBJECTS DBA_OBJECTS	Information about the objects. For table information such as creation timestamp and modification date, query this view. The OBJECT_TYPE column shows the type of the

USER_OBJECTS	object, such as table, index, trigger, etc.
ALL_EXTENTS DBA_EXTENTS USER_EXTENTS	Information about the extents allocated to the table. Shows the tablespace, data file, number of blocks, extent size, etc.

1.5. The Structure of a Row

Objective: Describe the structure of a row.

Oracle stores data in the form of rows. Rows are stored in blocks. You define the size of the block when you create the database, but you can override the size when you create tablespaces. If the entire row can be inserted into a block, the row is stored as a row piece. If the row to be stored is bigger than the block size, the row is stored using multiple row pieces. The same is true for rows that grow beyond the free space available in a block during updates.

A row piece contains a maximum of only 255 columns. If there is data beyond the 255th column in a row, the row is stored as multiple row pieces, a practice known as intra-block chaining. Intra-block chaining does not affect IO performance.

A row piece has two parts—a row header and column data. A row header is about 3 bytes for a row piece that is fully contained in a block.

The row header includes information about the row piece such as the columns, whether any rows are chained, and whether any cluster keys are present. After the row header is the column data. Column data has two pieces—length and data. The column length occupies 1 byte for data less than 251 bytes and 3 bytes for data over 250 bytes. To conserve space, Oracle does not store null values; only the column length is marked as zero. If the NULL columns are toward the end of the row, Oracle does not even store the column length.

1.6. Using ROWID

[redacted]

ROWID uniquely identifies each row of the table. ROWID is a pseudo-column in all tables that is not implicitly selected—you must specify ROWID in the query. The ROWID is an 18-byte structure that stores the physical location of the row. Since ROWIDs contain the exact block ID where the row is located, using ROWID is the fastest way to access a row. There are two categories of ROWIDs:

Physical Identifies each row of a table, partition, subpartition, or cluster

Logical Identifies the rows of an Index Organized Table (IOT—discussed later in this lesson)

Unless explicitly specified, ROWID in this lesson means a physical ROWID.

There are two formats for the ROWID:

Extended This format uses a base-64 encoding scheme to display the ROWID consisting of the characters A–Z, a–z, 0–9, +, and –. The ROWID is an 18-character representation that is stored in 10 bytes. The format is OOOOOOFFFBBBBBBRRR:

- OOOOOO is the object number.
- FFF is the relative data file number where the block is located; the file number is relative to the tablespace.
- BBBBBB is the block ID where the row is located.
- RRR is the row in the block.

```
SQL> SELECT ROWID, ORDER_NUM
  2  FROM ORDERS;
ROWID                  ORDER_NUM
---------------------- ---------
AAAFqsAADAAAAfTAAA     5934343
AAAFqsAADAAAAfTAAB     343433
```

Restricted This format is the pre-Oracle8 format, carried forward for compatibility. The restricted format is BBBBBBB.RRRR.FFFF (in base-16 or hexadecimal format):

- BBBBBBB is the data block.
- RRRR is the row number.
- FFFF is the data file.

```
SQL> SELECT DBMS_ROWID.ROWID_TO_RESTRICTED(ROWID,0),
  2  ORDER_NUM
  3  FROM ORDERS;
DBMS_ROWID.ROWID_T ORDER_NUM
-------------------------- ---------
000007D3.0000.0003         5934343
000007D3.0001.0003         343433
```

DBMS_ROWID

You can use the DBMS_ROWID package to read and convert the ROWID information. This package has several useful functions that you can use to convert the ROWID between extended and restricted formats. You use the function ROWID_TO_RESTRICTED to convert the extended ROWID format to restricted format. The two parameters to this function are the extended ROWID and the conversion type. Conversion type is an integer: 0 to return the ROWID in an internal format, and 1 to return it in a character string.

Oracle7 used restricted ROWID format, and Oracle8 and later versions use the extend ROWID format. If the database is upgraded from Oracle7 to Oracle8i, and there are some tables with a ROWID defined as the type of an explicit column in the table, you can convert the restricted ROWID to extended format by using the function ROWID_TO_EXTENDED. There are four parameters to this function: the old ROWID, the object owner, the object name, and the conversion type. The object owner and object name parameters are optional.

```
SQL> SELECT ROWID,
  2    DBMS_ROWID.ROWID_TO_EXTENDED(ROWID,
  3    NULL, NULL, 0) EXT_ROWID,
  4    DBMS_ROWID.ROWID_TO_RESTRICTED(ROWID,1) RES_ROWID
  5    FROM ORDERS
SQL> /
ROWID              EXT_ROWID          RES_ROWID
------------------ ------------------ ------------------
AAAFqsAADAAAAfTAAA AAAFqsAADAAAAfTAAA 000007D3.0000.0003
AAAFqsAADAAAAfTAAB AAAFqsAADAAAAfTAAB 000007D3.0001.0003
```

The ROWID_TO_VERIFY function takes the same parameters as the ROWID_TO_EXTENDED function. This function verifies whether a restricted format ROWID can be converted to extended format by using the ROWID_TO_EXTENDED function. If the ROWID can be converted, it returns 0; otherwise, it returns 1.

2. Managing Indexes

Objective: Describe the different types of indexes and their uses.

Indexes are used to access data more quickly than reading the whole table, and they reduce disk I/O considerably when the queries use the available indexes. As with tables, you can

specify storage parameters for indexes, create partitioned indexes, and analyze the index to verify structure and collect statistics. You can create any number of indexes on a table. A column can be part of multiple indexes, and you can specify as many as 30 columns in an index. When you specify more than one column, the index is known as a composite index. You can have more than one index with the same index columns, but in a different order.

You can create and drop indexes without affecting the base data of the table — indexes and table data are independent. Oracle maintains the indexes automatically: when new rows are added to the table, updated, or deleted, Oracle updates the corresponding indexes. You can create the following types of indexes:

Bitmap A *bitmap index* does not repeatedly store the index column values. Each value is treated as a key, and a bit is set for the corresponding ROWIDs. Bitmap indexes are suitable for columns with low cardinality, such as the SEX column in an EMPLOYEE table, in which the possible values are M or F. The *cardinality* is the number of distinct column values in a column. In the EMPLOYEE table example, the cardinality of the SEX column is 2. You cannot create unique or reverse key bitmap indexes.

b-tree This type of index is the default. You create the index by using the b-tree algorithm. The *b-tree* includes nodes with the index column values and the ROWID of the row. The ROWIDs identify the rows in the table. You can create the following types of b-tree indexes:

Non-unique This is the default b-tree index; the index column values are not unique.

Unique You create this type of b-tree index by specifying the UNIQUE keyword: each column value entry of the index is unique. Oracle guarantees that the combination of all index column values in the composite index is unique. Oracle returns an error if you try to insert two rows with the same index column values.

Reverse key To specify the *reverse key index* you use the REVERSE keyword. The bytes of each column indexed are reversed, but the column order is retained. For example, if column ORDER_NUM has value 54321, Oracle reverses the bytes to 12345 and then adds the column to the index. You can use this type of indexing for unique indexes when inserts to the table are always in the ascending order of the indexed columns. This type of indexing helps to distribute the adjacent valued columns to different leaf blocks of the index and, as a result, improve performance by retrieving fewer index blocks. *Leaf blocks* are the blocks at the lowest level of the b-tree.

Function-based You can create the *function-based index* on columns with expressions. For example, creating an index on the SUBSTR(EMPID, 1,2) can speed up the queries using SUBSTR(EMPID, 1, 2) in the WHERE clause.

TIP: Oracle does not include the rows with NULL values in the index columns when storing the b-tree index. Bitmap indexes store NULL values.

2.1. Creating Indexes

Objective: Create various types of indexes.

The CREATE INDEX statement creates a non-unique b-tree index on the columns specified. You must specify a name for the index and the table name on which the index should be built. For example, to create an index on the ORDER_DATE column of the ORDERS table, specify the following:

```
CREATE INDEX IND1_ORDERS
ON ORDERS (ORDER_DATE);
```

To create a unique index, you must specify the keyword UNIQUE immediately after CREATE. For example:

```
CREATE UNIQUE INDEX IND2_ORDERS
ON ORDERS (ORDER_NUM);
```

To create a bitmap index, you must specify the keyword BITMAP immediately after CREATE. Bitmap indexes cannot be unique. For example:

```
CREATE BITMAP INDEX IND3_ORDERS
ON ORDERS (STATUS);
```

Specifying Storage

If you do not specify the TABLESPACE clause in the CREATE INDEX statement, Oracle creates the index in the default tablespace of the user. If you don't specify the STORAGE clause, Oracle inherits the default storage parameters defined for the tablespace. All the storage parameters discussed in the "Managing Tables" section are applicable to indexes and have the same meaning except for PCTUSED. You cannot specify PCTUSED for indexes. Keep the INITRANS for the index more than that specified for the corresponding table, because the index blocks can hold a larger number of rows than a table.

Here is an example of creating an index and specifying the storage:

```
CREATE UNIQUE INDEX IND2_ORDERS
ON ORDERS (ORDER_NUM)
TABLESPACE USER_INDEX
```

```
PCTFREE 25
INITRANS 2
MAXTRANS 255
STORAGE (INITIAL 128K NEXT 128K PCTINCREASE 0
        MINEXTENTS 1 MAXEXTENTS 100
        FREELISTS 1 FREELIST GROUPS 1
        BUFFER_POOL KEEP);
```

When creating indexes for a table with rows, Oracle writes the data blocks with index values up to PCTFREE. The free space reserved by specifying PCTFREE is used when inserting into the table a new row (or updating a row that changes the corresponding index key column value) that needs to be placed between two index key values of the leaf node. If no free space is available in the block, Oracle uses a new block. If many new rows are inserted into the table, keep the PCTFREE of the index high.

Using Other Create Clauses

You can use the NOLOGGING clause to specify that information is not written to the redo log files, which speeds index creation. The default is LOGGING.

You can also collect statistics about the index while creating the index by specifying the COMPUTE STATISITCS clause. Using this clause avoids another ANALYZE on the index later.

The ONLINE clause specifies that the table will be available for DML operations when the index is built.

If data is loaded to the table in the order of an index, you can specify the NOSORT clause. Oracle does not sort the rows, but if the data is not sorted, Oracle returns an error. Specifying this clause saves time and temporary space.

For multicolumn indexes, eliminating the repeating key columns can save storage space. Specify the COMPRESS clause when creating the index. NOCOMPRESS is the default. This clause can be used only with non-partitioned indexes. Index performance may be affected when using this clause.

Specify PARALLEL to create the index using multiple server processes. NOPARALLEL is the default.

The following is an example of creating an index by specifying some of the miscellaneous clauses:

```
SQL> CREATE INDEX IND5_ORDERS ON ORDERS
  2    (ORDER_NUM, ORDER_DATE)
  3    TABLESPACE INDX
```

```
  4  NOLOGGING
  5  NOSORT
  6  COMPRESS
  7  SORT
  8  COMPUTE STATISTICS;
Index created.
SQL>
```

Partitioning

As with tables, you can partition indexes for better manageability and performance. Partitioned tables can have partitioned and/or non-partitioned indexes, and partitioned indexes can be created on partitioned or nonpartitioned tables. When all the index partitions correspond to the table partitions (equipartitioning), the index is called a local partitioned index.

Specifying the LOCAL keyword creates local indexes. For local indexes, Oracle maintains the index partition keys automatically, in synch with the table partition. A global partitioned index specifies different partition range values; the partition column values need not belong to a single table partition. Specifying the GLOBAL keyword creates global indexes.

You can create four types of partitioned indexes:

Local prefixed Local index with leading columns (leftmost column in index) in the order of the partition key.

Local non-prefixed Partition key columns are not leading columns, but the index is local.

Global prefixed Global index, with leading columns in the order of the partition key.

Global non-prefixed Global index, with leading columns not in the partition key order.

Warning: Bitmap indexes created on partitioned tables must be local. You cannot create a partitioned bitmap index on a non-partitioned table.

Reverse Key Indexes

Specifying the REVERSE keyword creates a reverse key index. Reverse key indexes improve performance of certain OLTP (Online Transaction Processing) applications using the parallel server. The following example creates a reverse key index on the ORDER_NUM and ORDER_DATE column of the ORDERS table.

```
CREATE UNIQUE INDEX IND2_ORDERS
```

```
ON ORDERS (ORDER_DATE, ORDER_NUM)
TABLESPACE USER_INDEX
REVERSE;
```

Function-Based Indexes

Function-based indexes are created as regular b-tree or bitmap indexes.

Specify the expression or function when creating the index. Oracle precalculates the value of the expression and creates the index. For example, to create a function based on SUBSTR(PRODUCT_ID,1,2), use the following:

```
CREATE INDEX IND4_ORDERS
ON ORDERS (SUBSTR(PRODUCT_ID,1,2))
TABLESPACE USER_INDEX;
```

To use the function-based index, you must set the instance initialization parameter QUERY_REWRITE_ENABLED to TRUE and the parameter QUERY_REWRITE_INTEGRITY to TRUSTED. Also, set the COMPATIBLE parameter to 8.1.0 or higher. A query can use this index if its WHERE clause specifies a condition by using SUBSTR(PRODUCT_ID,1,2), as in the following example:

```
SELECT * FROM ORDERS
WHERE SUBSTR(PRODUCT_ID,1,2) = 'BT';
```

TIP: You must gather statistics for function-based indexes for the cost-based optimizer to use the index. The rule-based optimizer does not use function-based indexes.

Index Organized Tables

You can store index and table data together in a structure known as an *Index Organized Table* (IOT). IOTs are suitable for tables in which the data access is mostly through its primary key, such as look-up tables, which have a code and a description. An IOT is a b-tree index, and instead of storing the ROWID of the table where the row belongs, the entire row is stored as part of the index. You can build additional indexes on the columns of an IOT. You access the data in an IOT in the same way you access the data in a table.

Since the row is stored along with the b-tree index, there is no physical ROWID for each row. The primary key identifies the rows in an IOT. Oracle "guesses" the location of the row and assigns a logical ROWID for each row, which permits the creation of secondary indexes. You can partition an IOT, but the partition columns should be a subset of the primary key columns.

To build additional indexes on the IOT, Oracle uses a logical ROWID, which is derived from the primary key values of the IOT. The logical ROWID can include a guessed physical location of the row in the data files. This guessed location is not valid when a row is moved from one block to another.

If the logical ROWID does not include the guessed location of the ROWID, Oracle has to perform two index scans when using the secondary index. The logical ROWIDs can be stored in columns with the datatype UROWID.

To create an IOT, you use the CREATE TABLE command with the ORGANIZATION INDEX keyword. You must specify the primary key for the table when creating the table.

```
SQL> CREATE TABLE IOT_EXAMPLE (
  2     PK_COL1 NUMBER (4),
  3     PK_COL2 VARCHAR2 (10),
  4     NON_PK_COL1 VARCHAR2 (40),
  5     NON_PK_COL2 DATE,
  6     CONSTRAINT PK_IOT PRIMARY KEY
  7     (PK_COL1, PK_COL2))
  8     ORGANIZATION INDEX
  9     TABLESPACE INDX
  10    STORAGE (INITIAL 32K NEXT 32K PCTINCREASE 0);
Table created.
SQL>
```

2.2. Altering Indexes

Using the ALTER INDEX command, you can make the following alterations in an index:

- Change its STORAGE clause, except for the parameters INITIAL and MINEXTENTS
- Deallocate unused blocks
- Rebuild the index
- Coalesce leaf nodes
- Manually allocate extents
- Change the PARALLEL/NOPARALLEL, LOGGING/NOLOGGING clauses
- Modify partition storage parameters, rename partitions, drop partitions, and so on
- Specify the ENABLE/DISABLE clause to enable or disable function-based indexes

- Mark the index or index partition as UNUSABLE, thereby disabling the index or index partition
- Rename the index

Since the rules for changing the storage parameters, allocating extents, and deallocating extents are similar to those for altering a table, we will provide only some short examples here.

To change storage parameters, use the following:

```
ALTER INDEX SCOTT.IND1_ORDERS
STORAGE (NEXT 512K MAXEXTENTS UNLIMITED);
```

To allocate an extent, use the following:

```
ALTER INDEX SCOTT.IND1_ORDERS
ALLOCATE EXTENT SIZE 200K;
```

To deallocate unused blocks, use the following:

```
ALTER INDEX SCOTT.IND1_ORDERS
DEALLOCATE UNUSED KEEP 100K;
```

If you disable an index by specifying the UNUSABLE clause, you must rebuild the index to make it valid.

Rebuilding/Coalescing Indexes

Objective: Reorganize indexes

Over time, blocks in an index can become fragmented and leave behind free space in leaf blocks. You can compress these indexes and gain space that can be used for new leaf blocks. You can use the COALESCE clause to free up index leaf blocks within the same branch of the tree. The index storage parameters and tablespace values remain the same. Here is an example:

```
ALTER INDEX IND1_ORDERS COALESCE;
```

If you want to re-create the index on a different tablespace or specify different storage parameters and free up leaf blocks, you can use the REBUILD clause. When you rebuild an index, Oracle drops the original index when the rebuild is complete. (A new index is created even if you do not change the tablespace or storage.) Users can access the index if you specify the ONLINE parameter. Optionally, you can collect statistics while rebuilding the index by using the COMPUTE STATISTICS parameter, and you can specify NOLOGGING so that redo log entries are not generated. You can also specify REVERSE or NOREVERSE to convert a normal index to a reverse key index or vice versa.

The following example moves the index to a new tablespace, collecting the statistics while rebuilding, and makes the table available for insert/update/delete operations by specifying the ONLINE clause.

```
ALTER INDEX IND1_ORDERS REBUILD
TABLESPACE NEW_INDEX_TS
STORAGE (INITIAL 25M NEXT 5M PCTINCREASE 0)
PCTFREE 20 INITRANS 4
COMPUTE STATISTICS
ONLINE NOLOGGING;
```

Dropping Indexes

Objective: Learn how to drop indexes

You can drop indexes using the DROP INDEX command. Oracle frees up all the space used by the index when the index is dropped. When a table is dropped, the indexes built on the table are automatically dropped. For example:

```
DROP INDEX SCOTT.IND5_ORDERS;
```

You cannot drop the indexes used to enforce the uniqueness or primary key of a table. Such indexes can be dropped only after disabling the primary and unique keys.

Analyzing Indexes

As you can with tables, you can analyze indexes to validate their structure (to find block corruption) and to collect statistics. You cannot use the ANALYZE command on an index with the LIST CHAINED ROWS clause. You can use the COMPUTE or ESTIMATE clause when collecting statistics. Here are some examples.

To validate the structure of an index, use the following:

```
ALTER INDEX IND5_ORDERS VALIDATE STRUCTURE;
```

To collect statistics by sampling 40 percent of the entries in the index, use the following:

```
ALTER INDEX IND5_ORDERS ESTIMATE STATISTICS
SAMPLE 40 PERCENT;
```

To delete statistics, use the following:

```
ALTER INDEX IND5_ORDERS DELETE STATISTICS;
```

Monitoring Index Usage

Objective: Monitor the usage of an index.

Oracle9i provides a method for detecting whether an index is used. You can drop unused indexes from the database to free up space and resources. Each index causes an overhead when DML statements are performed on the table. To enable index monitoring, you use the MONITORING USAGE clause of the ALTER INDEX statement.

```
ALTER INDEX <index name> MONITORING USAGE;
```

The V$OBJECT_USAGE view contains information about index usage. If the index is used after the monitoring begins, the USED column will have a value of YES. The following example illustrates index monitoring. It enables monitoring of index PK_DEPT.

```
SQL> alter index pk_dept monitoring usage;

Index altered.

SQL> select * from v$object_usage;

INDEX_NAME      TABLE_NAME      MON  USE  START_MONITORING
      END_MONITORING
----------      ----------      ---  ---  ------------------
      -------------
PK_DEPT         DEPT            YES  NO   11/23/2001 23:54:05
SQL>
```

The START_MONITORING column has the timestamp of when the monitoring began. Each time you start monitoring, this timestamp is reset. Using the NOMONITORING clause can stop the monitoring of the index.

```
ALTER INDEX <index name> NOMONITORING USAGE;
```

Let's now query the DEPT table and see how the USED column value changes.

```
SQL> select /*+ index (dept pk_dept) */ * from dept
  2     where deptno = 10;


DEPTNO      DNAME           LOC
----------  ------------    -------------
10          ACCOUNTING      NEW YORK
```

```
SQL> alter index pk_dept nomonitoring usage;

Index altered.

SQL> select * from v$object_usage;

INDEX_NAME      TABLE_NAME      MON  USE  START_MONITORING
      END_MONITORING
----------      ----------      ---  ---  -------------------
      -------------------
PK_DEPT         DEPT            NO   YES  11/23/2001    23:54:05
      11/24/2001 23:54:05

SQL>
```

2.3. Querying Index Information

Objective: Get index information from the data dictionary.

Several data dictionary views are available to query information about indexes. This section covers certain views and their columns that you should be familiar with.

DBA_INDEXES

The DBA_INDEXES, USER_INDEXES, and ALL_INDEXES views are the primary views you can use to query for information about indexes (IND is a synonym for USER_INDEXES). The views have the following information (the columns that can be used in the query are provided in parentheses):

- Identity (OWNER, INDEX_NAME, INDEX_TYPE, TABLE_OWNER, TABLE_NAME, TABLE_TYPE)
- Storage (TABLESPACE_NAME, PCT_FREE, PCT_USED, INI_TRANS, MAX_TRANS, INITIAL_EXTENT, NEXT_EXTENT, MAX_EXTENTS, MIN_EXTENTS, PCT_INCREASE, FREELISTS, FREELIST_GROUPS, BUFFER_POOL)
- Statistics (BLEVEL, LEAF_BLOCKS, DISTINCT_KEYS, AVG_LEAF_BLOCKS_PER_KEY, AVG_DATA_BLOCKS_PER_KEY, NUM_ROWS, SAMPLE_SIZE, LAST_ANALYZED)
- Miscellaneous create options (UNIQUENESS, LOGGING, DEGREE, CACHE, PARTITIONED)

DBA_IND_COLUMNS

Use the DBA_IND_COLUMNS, USER_IND_COLUMNS, and ALL_IND_COLUMNS views to display information about the columns in an index. The following information can be queried:

- Identity (INDEX_OWNER, INDEX_NAME, TABLE_OWNER, TABLE_NAME, COLUMN_NAME)
- Column characteristics (COLUMN_LENGTH, COLUMN_POSITION, DESCEND)

Table 4 lists other dictionary views that show information about the indexes in the database.

TABLE 4. Dictionary Views with Index Information.

View Name	Contents
ALL_IND_PARTITIONS DBA_IND_PARTITIONS USER_IND_PARTITIONS	Index partitioning information, storage parameters, and partition-level statistics gathered
ALL_IND_SUBPARTITIONS DBA_IND_SUBPARTITIONS USER_IND_SUBPARTITIONS	Sub-partition information for composite index partitions in the database
ALL_IND_EXPRESSIONS DBA_IND_EXPRESSIONS USER_IND_EXPRESSIONS	Columns or expressions used to create the function-based index
INDEX_STATS	Statistical information from the ANALYZE INDEX VALIDATE STRUCTURE command

3. Managing Constraints

Constraints are created in the database to enforce a business rule and to specify relationships between various tables. You can also enforce business rules by using database triggers and application code. *Integrity constraints* prevent bad data from being entered into the database.

Oracle allows you to create five types of integrity constraints:

NOT NULL Prevents NULL values from being entered into the column. These types of constraints are defined on a single column.

CHECK Checks whether the condition specified in the constraint is satisfied.

UNIQUE Ensures that there are no duplicate values for the column(s) specified. Every value or set of values is unique within the table.

PRIMARY KEY Uniquely identifies each row of the table. Prevents NULL values. A table can have only one primary key constraint.

FOREIGN KEY Establishes a parent-child relationship between tables by using common columns.

3.1. Creating Constraints

Objective: Implement data integrity constraints.

To create constraints, you use the CREATE TABLE or ALTER TABLE statements. You can specify the constraint definition at the column level if the constraint is defined on a single column. You define multiple column constraints at the table level; specify the columns in parentheses separated by a comma. If you do not provide a name for the constraints, Oracle assigns a system-generated name. To provide a name for the constraint, specify the keyword CONSTRAINT followed by the constraint name. In this section, we discuss the rules for each constraint type and give you some examples of creating constraints.

NOT NULL

NOT NULL constraints have the following characteristics:

- You define the constraint at the column level.
- Use CREATE TABLE to define constraints when creating the table.

  The following example shows a named constraint on the ORDER_NUM column; for ORDER_DATE, Oracle generates a name.

```
CREATE TABLE ORDERS (
ORDER_NUM NUMBER (4) CONSTRAINT NN_ORDER_NUM NOT NULL,
ORDER_DATE DATE NOT NULL,
PRODUCT_ID)
```

- Use ALTER TABLE MODIFY to add or remove a NOT NULL constraint on the columns of an existing table.

  The following code shows examples of removing a constraint and adding a constraint.

```
ALTER TABLE ORDERS MODIFY ORDER_DATE NULL;
ALTER TABLE ORDERS MODIFY PRODUCT_ID NOT NULL;
```

CHECK

CHECK constraints have the following characteristics:

- They can be defined at the column level or table level.
- The condition specified in the CHECK clause should evaluate to a Boolean result and can refer to values in other columns of the same row; the condition cannot use queries.
- Environment functions such as SYSDATE, USER, USERENV, UID, and pseudo-columns such as ROWNUM, CURRVAL, NEXTVAL, or LEVEL cannot be used to evaluate the check condition.
- One column can have more than one CHECK constraint defined. The column can have a NULL value.
- They can be created using CREATE TABLE or ALTER TABLE.

```
CREATE TABLE BONUS (
EMP_ID VARCHAR2 (40) NOT NULL,
SALARY NUMBER (9,2),
BONUS NUMBER (9,2),
CONSTRAINT CK_BONUS CHECK (BONUS > 0));


ALTER TABLE BONUS
ADD CONSTRAINT CK_BONUS2 CHECK (BONUS < SALARY);
```

UNIQUE

UNIQUE constraints have the following characteristics:

- They can be defined at the column level for single-column unique keys. For a multiple-column unique key (composite key—the maximum number of columns specified can be 32), the constraint should be defined at the table level.
- Oracle creates a unique index on the unique key columns to enforce uniqueness. If a unique index or a non-unique index already exists on the table with the same columns in the index, Oracle uses the existing index. To use the existing non-unique index, the table must not contain any duplicate keys.
- Unique constraints allow NULL values in the constraint columns.
- Storage can be specified for the implicit index created when creating the key. If no storage is specified, the index is created on the default tablespace with the default storage parameters of the tablespace. You can specify the LOGGING and NOSORT clauses, as you would when creating an index. The index created can be a local or a global partitioned index. The index will have the same name as the unique constraint. Following are two examples. The first one defines a unique constraint with two columns and specifies the storage parameters for the index. The second example adds a new column to the EMP table and creates a unique key at the column level.

```
ALTER TABLE BONUS
ADD CONSTRAINT UQ_EMP_ID UNIQUE (DEPT, EMP_ID)
USING INDEX TABLESPACE INDX
STORAGE (INITIAL 32K NEXT 32K PCTINCREASE 0);


ALTER TABLE EMP ADD
SSN VARCHAR2 (11) CONSTRAINT UQ_SSN UNIQUE;
```

PRIMARY KEY

PRIMARY KEY constraints have the following characteristics:

- All characteristics of the UNIQUE key are applicable except that NULL values are not allowed in the primary key columns.
- A table can have only one primary key.
- Oracle creates a unique index and NOT NULL constraints for each column in the key. Oracle can use an existing index if all the columns of the primary key are in the index. The following example defines a primary key when creating the table. Storage parameters are specified for both the table and the primary key index.

```
CREATE TABLE EMPLOYEE (
DEPT_NO VARCHAR2 (2),
EMP_ID NUMBER (4),
NAME VARCHAR2 (20) NOT NULL,
SSN VARCHAR2 (11),
SALARY NUMBER (9,2) CHECK (SALARY > 0),
CONSTRAINT PK_EMPLOYEE PRIMARY KEY (DEPT_NO, EMP_ID)
USING INDEX TABLESPACE INDX
STORAGE (INITIAL 64K NEXT 64K)
NOLOGGING,
CONSTRAINT UQ_SSN UNIQUE (SSN)
USING INDEX TABLESPACE INDX)
TABLESPACE USERS
STORAGE (INITIAL 128K NEXT 64K);
```

- Indexes created to enforce unique keys and primary keys can be managed as any other index. However, these indexes cannot be dropped explicitly.

Warning : You cannot drop indexes created to enforce the primary key or unique constraints.

FOREIGN KEY

The foreign key is the column or columns in the table (child table) in which the constraint is created; the referenced key is the primary key, the unique key column, or columns in the table (parent table) that is referenced by the constraint. The following rules are applicable to foreign key constraints:

- You can define a foreign key constraint at the column level or table level. Define multiple-column foreign keys at the table level.
- The foreign key column(s) and referenced key column(s) can be in the same table (self-referential integrity constraint).
- NULL values are allowed in the foreign key columns. The following is an example of creating a foreign key constraint on the COUNTRY_CODE and STATE_CODE columns of the CITY table, which refers to the COUNTRY_CODE and STATE_CODE columns of the STATE table (the composite primary key of the STATE table).

```
ALTER TABLE CITY ADD CONSTRAINT FK_STATE
FOREIGN KEY (COUNTRY_CODE, STATE_CODE)
REFERENCES STATE (COUNTRY_CODE, STATE_CODE);
```

- The ON DELETE clause specifies the action to be taken when a row in the parent table is deleted and child rows exist with the deleted parent primary key. You can delete the child rows (CASCADE) or set the foreign key column values to NULL (SET NULL). If you omit this clause, Oracle will not allow you to delete from the parent table if child records exist. You must delete the child rows first and then the parent row. Following are two examples of specifying the delete action in a foreign key.

```
ALTER TABLE CITY ADD CONSTRAINT FK_STATE
FOREIGN KEY (COUNTRY_CODE, STATE_CODE)
REFERENCES STATE (COUNTRY_CODE, STATE_CODE)
ON DELETE CASCADE;
```

```
ALTER TABLE CITY ADD CONSTRAINT FK_STATE
FOREIGN KEY (COUNTRY_CODE, STATE_CODE)
REFERENCES STATE (COUNTRY_CODE, STATE_CODE)
ON DELETE SET NULL;
```

Creating Disabled Constraints

When you create a constraint, it is enabled automatically. You can create a disabled constraint by specifying the DISABLED keyword after the constraint definition. For example:

```
ALTER TABLE CITY ADD CONSTRAINT FK_STATE
FOREIGN KEY (COUNTRY_CODE, STATE_CODE)
REFERENCES STATE (COUNTRY_CODE, STATE_CODE) DISABLE;


ALTER TABLE BONUS
ADD CONSTRAINT CK_BONUS CHECK (BONUS > 0) DISABLE;
```

Dropping Constraints

To drop constraints, you use ALTER TABLE. You can drop any constraint by specifying the constraint name.

```
ALTER TABLE BONUS DROP CONSTRAINT CK_BONUS2;
```

To drop unique key constraints with referenced foreign keys, specify the CASCADE clause to drop the foreign key constraints and the unique constraint. Specify the unique key columns(s). For example:

```
ALTER TABLE EMPLOYEE DROP UNIQUE (EMP_ID) CASCADE;
```

To drop primary key constraints with referenced foreign key constraints, use the CASCADE clause to drop all foreign key constraints and then the primary key.

```
ALTER TABLE BONUS DROP PRIMARY KEY CASCADE;
```

3.2. Enabling and Disabling Constraints

Objective: Maintain integrity constraints.

When you create a constraint, the constraint is automatically enabled (unless you specify the DISABLE clause). You can disable a constraint by using the DISABLE clause of the ALTER TABLE statement. When you disable the UNIQUE or PRIMARY KEY constraints, Oracle drops the associated unique index. When you re-enable these constraints, Oracle builds the index.

You can disable any constraint by specifying the clause DISABLE CONSTRAINT followed by the constraint name. Specifying UNIQUE and the column name(s) can disable unique keys, and specifying PRIMARY KEY can disable the table's primary key. You cannot disable

a primary key or a unique key if foreign keys that are enabled reference it. To disable all the referenced foreign keys and the primary or unique key, specify CASCADE.

Following are three examples that illustrate disabling.

```
ALTER TABLE BONUS DISABLE CONSTRAINT CK_BONUS;

ALTER TABLE EMPLOYEE DISABLE CONSTRAINT UQ_EMPLOYEE;

ALTER TABLE STATE DISABLE PRIMARY KEY CASCADE;
```

Using the ENABLE clause of the ALTER TABLE statement enables a constraint. When you enable a disabled unique or primary key, Oracle creates an index if an index with the unique or primary key columns (prefixed) does not already exist. You can specify storage for the unique or primary key when enabling these constraints. For example:

```
ALTER TABLE STATE ENABLE PRIMARY KEY USING INDEX

TABLESPACE USER_INDEX STORAGE (INITIAL 2M NEXT 2M);
```

You can use the EXCEPTIONS INTO clause to find the rows that violate a referential integrity or uniqueness condition. The ROWIDs of the invalid rows are inserted into a table. You can specify the name of the table in which you want the ROWIDs to be saved; by default, Oracle looks for the table named EXCEPTIONS. You can create the table using the script utlexcpt.sql supplied from Oracle, located in the rdbms/admin directory of the software installation. The structure of the table is as follows:

```
SQL> @c:\oracle\ora90\rdbms\admin\utlexcpt.sql

Table created.

SQL> desc exceptions

Name          Null?  Type

----------------------- -------- -------------

ROWID                  ROWID

OWNER                  VARCHAR2(30)

TABLE_NAME             VARCHAR2(30)

CONSTRAINT             VARCHAR2(30)
```

The following example enables the primary key constraint and inserts the ROWIDs of the bad rows into the EXCEPTIONS table:

```
ALTER TABLE STATE ENABLE PRIMARY KEY

EXCEPTIONS INTO EXCEPTIONS;
```

You can also use the MODIFY CONSTRAINT clause of the ALTER TABLE statement to enable/disable constraints. Specify the constraint name followed by the MODIFY CONSTRAINT keywords. Following are examples.

```
ALTER TABLE BONUS MODIFY CONSTRAINT CK_BONUS DISABLE;

ALTER TABLE STATE MODIFY CONSTRAINT PK_STATE

DISABLE CASCADE;

ALTER TABLE BONUS MODIFY CONSTRAINT CK_BONUS ENABLE;

ALTER TABLE STATE MODIFY CONSTRAINT PK_STATE USING INDEX

TABLESPACE USER_INDEX STORAGE (INITIAL 2M NEXT 2M) ENABLE;
```

Validated Constraints

You have seen how to enable and disable a constraint. ENABLE and DISABLE affect only future data that will be added/modified in the table. In contrast, the VALIDATE and NOVALIDATE keywords in the ALTER TABLE command act on the existing data. Therefore, a constraint can have four states:

ENABLE VALIDATE This is the default for the ENABLE clause. The existing data in the table is validated to verify that it conforms to the constraint.

ENABLE NOVALIDATE This constraint does not validate the existing data, but enables the constraint for future constraint checking.

DISABLE VALIDATE The constraint is disabled (any index used to enforce the constraint is dropped), but the constraint remains valid. No DML operation is allowed on the table because future changes cannot be verified.

DISABLE NOVALIDATE This is the default for the DISABLE clause. The constraint is disabled, and no checks are done on future or existing data.

Let's look at an example of how these clauses can be used. Say that you have a large data warehouse table, on which bulk data loads are performed every night. This table has a primary key enforced using a non-unique index — because Oracle does not drop the non-unique index when disabling the constraint. When you do batch loads, you can disable the primary key constraint as follows:

```
ALTER TABLE WH01 MODIFY CONSTRAINT PK_WH01

DISABLE NOVALIDATE;
```

After the batch load completes, you can enable the primary key as follows:

```
ALTER TABLE WH01 MODIFY CONSTRAINT PK_WH01

ENABLE NOVALIDATE;
```

TIP: Oracle does not allow any inserts/updates/deletes on a table with a DISABLE VALIDATE constraint. Changing the constraint status to DISABLE VALIDATE is a quick way to make a table read-only.

REAL WORLD SCENARIO

Creating a Table, Indexes, and Constraints for a Customer Maintenance Application

You, the DBA, must create the tables that are needed to manage a customer database. You are given the physical structure of the tables, the relationship between the tables, and the following information:

- Columns of the CUSTOMER_MASTER table and the type of data stored. CUST_ID is the unique identifier of the table—the primary key. This table contains customer name, e-mail address, date of birth, primary contact address type, and status flag.
- The CUSTOMER_ADDRESS table keeps the addresses of the customers. Each customer can have a maximum of four addresses—business1, business2, home1, and home2.
- CUSTOMER_REFERENCES table keeps information about the new customers introduced by a customer. This table simply keeps the customer ID of the referring customer and the new customer.
- Each table has record creation date, created user name, update date, and update username.

You decide to keep the tables and indexes in separate tablespaces and to use the uniform extent feature of the tablespace. This arrangement helps to manage the space on the tablespace more effectively. Data is kept in CUST_DATA tablespace, and indexes are maintained in the CUST_INDX tablespace. You create the tablespaces as follows:

```
CREATE TABLESPACE CUST_DATA DATAFILE
'C:\ORACLE\ORADATA\CUST_DATA01.DBF' SIZE 512K
AUTOEXTEND ON NEXT 128K MAXSIZE 2000K
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 64K
SEGMENT SPACE MANAGEMENT AUTO;
CREATE TABLESPACE CUST_INDX DATAFILE
'C:\ORACLE\ORADATA\CUST_INDX.DBF' SIZE 256K
AUTOEXTEND ON NEXT 128K MAXSIZE 2000K
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 32K
SEGMENT SPACE MANAGEMENT AUTO;
```

Now you need to create the CUSTOMER_MASTER table. The table needs to have a primary key CUST_ID and a unique key EMAIL. You create a non-unique index on the EMAIL column, which is used to enforce the UNIQUE key. You also want to create an index on the DOB column because each week the firm sends greetings to all customers who are celebrating

a birthday that week.

The check constraint on the ADD_TYPE ensures that no other values are inserted into the column.

```
CREATE TABLE CUSTOMER_MASTER (
CUST_ID VARCHAR2 (10),
CUST_NAME VARCHAR2 (30),
EMAIL VARCHAR2 (30),
DOB DATE,
ADD_TYPE CHAR (2) CONSTRAINT CK_ADD_TYPE
        CHECK (ADD_TYPE IN ('B1','B2','H1','H2')),
CRE_USER VARCHAR2 (5) DEFAULT USER,
CRE_TIME TIMESTAMP (3) DEFAULT SYSTIMESTAMP,
MOD_USER VARCHAR2 (5),
MOD_TIME TIMESTAMP (3),
CONSTRAINT PK_CUSTOMER_MASTER PRIMARY KEY (CUST_ID)
        USING INDEX TABLESPACE CUST_INDX)
TABLESPACE CUST_DATA;
CREATE INDEX CUST_EMAIL ON CUSTOMER_MASTER (EMAIL)
TABLESPACE CUST_INDX;
ALTER TABLE CUSTOMER_MASTER ADD CONSTRAINT UQ_CUST_EMAIL
UNIQUE (EMAIL) USING INDEX CUST_EMAIL;
```

Now you are ready to create the CUSTOMER_ADDRESSES table. You create the table first and then add the primary key and the foreign key.

You create the foreign key with an option to defer its checking until commit time.

```
CREATE TABLE CUSTOMER_ADDRESSES (
CUST_ID VARCHAR2 (10),
ADD_TYPE CHAR (2),
ADD_LINE1 VARCHAR2 (40) NOT NULL,
ADD_LINE2 VARCHAR2 (40),
CITY VARCHAR2 (40) NOT NULL,
STATE VARCHAR2 (2) NOT NULL,
ZIP NUMBER (5) NOT NULL)
TABLESPACE CUST_DATA;
ALTER TABLE CUSTOMER_ADDRESSES ADD CONSTRAINT
PK_CUST_ADDRESSES PRIMARY KEY (CUST_ID, ADD_TYPE)
USING INDEX TABLESPACE CUST_INDX;
ALTER TABLE CUSTOMER_ADDRESSES ADD CONSTRAINT
```

```
FK_CUST_ADDRESSES FOREIGN KEY (CUST_ID)
REFERENCES CUSTOMER_MASTER;
ALTER TABLE CUSTOMER_ADDRESSES ADD CONSTRAINT
CK_ADD_TYPE2 CHECK (ADD_TYPE IN ('B1','B2','H1','H2'));
```

You forgot, however, to enable the constraint deferrable clause and to delete records from CUSTOMER_ADDRESSES table when the row is deleted from CUSTOMER_MASTER table, so you do that now as follows:

```
ALTER TABLE CUSTOMER_ADDRESSES
DROP CONSTRAINT FK_CUST_ADDRESSES;
ALTER TABLE CUSTOMER_ADDRESSES ADD CONSTRAINT
FK_CUST_ADDRESSES FOREIGN KEY (CUST_ID)
REFERENCES CUSTOMER_MASTER
ON DELETE CASCADE DEFERRABLE INITIALLY IMMEDIATE;
```

Now you create the CUSTOMER_REFERENCES table. Since this table row never grows with updates, you set the PCTFREE of the table to 0.

```
CREATE TABLE CUSTOMER_REFERENCES (
CUST_ID VARCHAR2 (10) REFERENCES CUSTOMER_MASTER,
CUST_REF_ID VARCHAR2 (10) REFERENCES CUSTOMER_MASTER,
CRE_USER VARCHAR2 (5),
CRE_TIME TIMESTAMP (3) DEFAULT SYSTIMESTAMP,
MOD_USER VARCHAR2 (5) DEFAULT USER,
MOD_TIME TIMESTAMP (3),
CONSTRAINT PK_CUST_REFS PRIMARY KEY (CUST_ID, CUST_REF_ID))
TABLESPACE CUST_DATA
PCTFREE 0;
```

By reviewing the creating, you find that the CUSTOMER_ADDRESSES table does not have the created and modified user information and that the CUSTOMER_REFERENCES table has a DEFAULT value assigned to the wrong column. You fix these problems as follows:

```
ALTER TABLE CUSTOMER_ADDRESSES ADD (
CRE_USER VARCHAR2 (5) DEFAULT USER,
CRE_TIME TIMESTAMP (3) DEFAULT SYSTIMESTAMP,
MOD_USER VARCHAR2 (5),
MOD_TIME TIMESTAMP (3));
ALTER TABLE CUSTOMER_REFERENCES MODIFY
MOD_USER DEFAULT NULL;
ALTER TABLE CUSTOMER_REFERENCES MODIFY
CRE_USER DEFAULT USER;
```

Also, the primary key for the CUSTOMER_REFERENCES table did not specify a tablespace for the primary key index, so it was created in the default tablespace of the table. You correct this as follows:

```
SQL> SELECT TABLESPACE_NAME FROM DBA_INDEXES WHERE
  2    INDEX_NAME = 'PK_CUST_REFS';
TABLESPACE_NAME
------------------------------
CUST_DATA
SQL> ALTER INDEX PK_CUST_REFS REBUILD TABLESPACE CUST_INDX;
Index altered.
SQL> SELECT TABLESPACE_NAME FROM DBA_INDEXES WHERE
  2    INDEX_NAME = 'PK_CUST_REFS';
TABLESPACE_NAME
------------------------------
CUST_INDX
SQL>
```

Now you query the dictionary views to see the table information.

```
SQL> SELECT TABLE_NAME, TABLESPACE_NAME
  2    FROM USER_TABLES
  3    WHERE TABLE_NAME LIKE 'CUST%';
TABLE_NAME                     TABLESPACE_NAME
------------------------------ ------------------------
CUSTOMER_ADDRESSES             CUST_DATA
CUSTOMER_MASTER                CUST_DATA
CUSTOMER_REFERENCES            CUST_DATA
SQL> SELECT SEGMENT_NAME, SEGMENT_TYPE, TABLESPACE_NAME,
BYTES
  2  FROM DBA_SEGMENTS
  3  WHERE OWNER = 'CM'
  4  AND SEGMENT_NAME LIKE '%CUST%';
SEGMENT_NAME          SEGMENT_TYPE     TABLESPACE  BYTES
--------------------  ---------------  ----------  ----------
CUSTOMER_MASTER       TABLE            CUST_DATA   65536
CUSTOMER_ADDRESSES    TABLE            CUST_DATA   65536
CUSTOMER_REFERENCES   TABLE            CUST_DATA   65536
PK_CUSTOMER_MASTER    INDEX            CUST_INDX   32768
CUST_EMAIL            INDEX            CUST_INDX   32768
```

```
PK_CUST_ADDRESSES     INDEX            CUST_INDX   32768
PK_CUST_REFS          INDEX            CUST_INDX   65536
7 rows selected.
SQL>
SQL> SELECT INDEX_NAME, COLUMN_NAME, COLUMN_POSITION
  2   FROM USER_IND_COLUMNS
  3   WHERE INDEX_NAME LIKE '%CUST%'
  4   ORDER BY 1,3;
INDEX_NAME                  COLUMN_NAME    COLUMN_POSITION
--------------------------- -------------- ---------------
CUST_EMAIL                  EMAIL           1
PK_CUSTOMER_MASTER          CUST_ID         1
PK_CUST_ADDRESSES           CUST_ID         1
PK_CUST_ADDRESSES           ADD_TYPE        2
PK_CUST_REFS                CUST_ID         1
PK_CUST_REFS                CUST_REF_ID     2
6 rows selected.
SQL>
```

Next you query the dictionary views to see the constraint information.

Notice that the two foreign key constraints on CUSTOMER_REFERENCES table and the NOT NULL constraints in the CUSTOMER_ADDRESSES table have system generated names.

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME,
  2   R_CONSTRAINT_NAME
  3   FROM USER_CONSTRAINTS
  4   WHERE TABLE_NAME LIKE 'CUST%';
CONSTRAINT_NAME   C TABLE_NAME          R_CONSTRAINT_NAME
----------------- - ------------------- -----------------
SYS_C002792       C CUSTOMER_ADDRESSES
SYS_C002793       C CUSTOMER_ADDRESSES
SYS_C002794       C CUSTOMER_ADDRESSES
SYS_C002795       C CUSTOMER_ADDRESSES
PK_CUST_ADDRESSES P CUSTOMER_ADDRESSES
FK_CUST_ADDRESSES R CUSTOMER_ADDRESSES PK_CUSTOMER_MASTE
CK_ADD_TYPE2      C CUSTOMER_ADDRESSES
CK_ADD_TYPE       C CUSTOMER_MASTER
PK_CUSTOMER_MASTER P CUSTOMER_MASTER
```

```
UQ_CUST_EMAIL     U  CUSTOMER_MASTER
PK_CUST_REFS      P  CUSTOMER_REFERENCES
SYS_C002804       R CUSTOMER_REFERENCES  PK_CUSTOMER_MASTER
SYS_C002805       R  CUSTOMER_REFERENCES PK_CUSTOMER_MASTER
13 rows selected.
SQL>
SQL> SELECT CONSTRAINT_NAME, GENERATED, INDEX_NAME
  2   FROM USER_CONSTRAINTS
  3   WHERE TABLE_NAME LIKE 'CUST%';
CONSTRAINT_NAME     GENERATED          INDEX_NAME
------------------- -------------      -------------------
SYS_C002792         GENERATED NAME
SYS_C002793         GENERATED NAME
SYS_C002794         GENERATED NAME
SYS_C002795         GENERATED NAME
PK_CUST_ADDRESSES   USER NAME          PK_CUST_ADDRESSES
FK_CUST_ADDRESSES   USER NAME
CK_ADD_TYPE2        USER NAME
CK_ADD_TYPE         USER NAME
PK_CUSTOMER_MASTER  USER NAME          PK_CUSTOMER_MASTER
UQ_CUST_EMAIL       USER NAME          CUST_EMAIL
PK_CUST_REFS        USER NAME          PK_CUST_REFS
SYS_C002804         GENERATED NAME
SYS_C002805         GENERATED NAME
13 rows selected.
SQL>
```

3.3. Deferring Constraints Checks

By default, Oracle checks whether the data conforms to the constraint when the statement is executed. Oracle allows you to change this behavior if the constraint is created using the DEFERRABLE clause (NOT DEFERRABLE is the default). Oracle specifies that the transaction can set the constraint-checking behavior. INITIALLY IMMEDIATE specifies that the constraint be checked for conformance at the end of each SQL statement (this is the default).

INITIALLY DEFERRED specifies that the constraint be checked for conformance at the end of the transaction. You cannot change the DEFERRABLE status of a constraint using ALTER

TABLE MODIFY CONSTRAINT; you must drop and re-create the constraint, and you can change the INITIALLY [DEFERRED/IMMEDIATE] clause using ALTER TABLE.

If the constraint is DEFERRABLE, you can change the behavior by using the SET CONSTRAINTS command or by using the ALTER SESSION SET CONSTRAINT command. You can enable or disable deferred constraint checking by listing all the constraints or by specifying the ALL keyword.

You use the SET CONSTRAINTS command to set the constraint-checking behavior for the current transaction, and you use the ALTER SESSION command to set the constraint-checking behavior for the current session.

Let's look at an example. Create a primary key constraint on the CUSTOMER table and a foreign key constraint on the ORDERS table as DEFERRABLE. Although the constraints are created DEFERRABLE, they are not deferred because of the INITIALLY IMMEDIATE clause.

```
ALTER TABLE CUSTOMER ADD CONSTRAINT PK_CUST_ID
PRIMARY KEY (CUST_ID) DEFERRABLE
INITIALLY IMMEDIATE;
ALTER TABLE ORDERS ADD CONSTRAINT FK_CUST_ID
FOREIGN KEY (CUST_ID)
REFERENCES CUSTOMER (CUST_ID)
ON DELETE CASCADE DEFERRABLE;
```

If you try to add a row to the ORDERS table with a CUST_ID that is not available in the CUSTOMER table, Oracle returns an error immediately, even though you plan to add the CUSTOMER row soon. Since the constraints are verified for conformance at each SQL statement, you must insert the CUSTOMER row first and then insert the row to the ORDERS table. Because the constraints are defined as DEFERRABLE, you can change this behavior by using the following command:

```
SET CONSTRAINTS ALL DEFERRED;
```

Now, you can insert rows in these tables in any order. Oracle checks the constraint conformance only at commit time.

If you want deferred constraint checking as the default, create/modify the constraint by using INITIALLY DEFERRED. For example:

```
ALTER TABLE CUSTOMER MODIFY CONSTRAINT PK_CUST_ID
INITIALLY DEFERRED;
```

3.4. Querying Constraints Information

Objective: Obtain constraint information from the data dictionary.

You can query constraints, their columns, type, status, and so on from the dictionary, using the following views.

DBA_CONSTRAINTS

You can query the ALL_CONSTRAINTS, DBA_CONSTRAINTS, and USER_CONSTRAINTS views to get information about the constraints. The CONSTRAINT_TYPE column shows the type of constraint — C for check, P for primary key, U for unique key, and R for referential (foreign key); V and O are associated with views. For check constraints, the SEARCH_CONDITION column shows the check condition. NOT NULL constraints are listed in this view as CHECK constraints. NOT NULL constraint information can also be found in the NULLABLE column of the DBA_TAB_COLUMNS view. Here is a sample query to get the constraint information:

```
SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, DEFERRED,
  2  DEFERRABLE, STATUS
  3  FROM DBA_CONSTRAINTS
  4  WHERE TABLE_NAME = 'ORDERS';

CONSTRAINT_NAME      C  DEFERRED      DEFERRABLE       STATUS
-----------------    -  ---------     --------------   --------
CK_QUANTITY          C  IMMEDIATE     NOT DEFERRABLE   DISABLED
PK_ORDERS            P  DEFERRED      DEFERRABLE       ENABLED
SQL>
```

DBA_CONS_COLUMNS

The ALL_CONS_COLUMNS, DBA_CONS_COLUMNS, and USER_CONS_COLUMNS views show the columns associated with the constraints.

```
SQL> SELECT CONSTRAINT_NAME, COLUMN_NAME, POSITION
  2  FROM DBA_CONS_COLUMNS
  3  WHERE TABLE_NAME = 'ORDERS';

CONSTRAINT_NAME      COLUMN_NAME     POSITION
-----------------------------  --------------  ----------
```

```
CK_QUANTITY          QUANTITY
PK_ORDERS            ORDER_NUM        2
PK_ORDERS            ORDER_DATE       1
SQL>
```

4. Summary

This lesson discussed the various options available for creating tables, indexes, and constraints. You create tables using the CREATE TABLE command. By default, the table is created in the current schema. To create the table in another schema, you qualify the table with the schema name. You can create storage parameters when creating the table. The storage parameters that specify the extent sizes are INITIAL, NEXT, and PCTINCREASE. Once the table is created, you cannot change the INITIAL and MINEXTENTS parameters. PCTFREE controls the free space in the data block.

Partitioning the tables lets you manage large tables more easily and results in better query performance. Partitioning is breaking a large table into smaller, more manageable pieces. Four partitioning methods are available: range, hash, list and composite. You can also create indexes on partitioned tables. The indexes can be equipartitioned, which results in what is also known as a local index, meaning index partitions will have the same partition keys and number of partitions as the partitioned table. You can create partitioned indexes on partitioned tables or non-partitioned tables. Similarly, partitioned tables can have partitioned local, partitioned global, or nonpartitioned indexes.

You can alter tables and indexes to deallocate extents or unused blocks. You use the DEALLOCATE clause of the ALTER TABLE statement to release the free blocks that are above the high-water mark (HWM). You can also manually allocate space to a table or an index by using the ALLOCATE EXTENT clause. You move or reorganize tables can be using the MOVE clause. You can specify new tablespace and storage parameters.

You can use the ANALYZE command on tables and indexes to validate the structure and to identify corrupt blocks. You can also use the ANALYZE command to collect statistics and to find and fix the chained rows in a table. The ROWID of the table is an 18-character representation of the physical location of the row. You can use the DBMS_ROWID package to convert ROWIDs between restricted and extended formats. You can query information on tables from DBA_TABLES, DBA_TAB_COLUMNS, DBA_TAB_PARTITIONS, and so on.

You can create indexes as b-tree or bitmap. Bitmap indexes save storage space for low cardinality columns. You can create reverse key or function-based indexes. An Index Organized Table (IOT) stores the index and row data in the b-tree structure. Specify tablespace and storage when creating indexes. When you create indexes ONLINE, the table is available for insert/update/delete operations during indexing. You can use the REBUILD clause of the ALTER INDEX command to move the index to a different tablespace or to

reorganize the index. You can also change a reverse key index to a normal index and vice versa. You can monitor index usage and then drop unused indexes from the database to save resources.

You create constraints on tables to enforce business rules. There are five types of constraints: not null, check, unique, primary key, and foreign key. You can create the constraints to check conformance at each SQL statement or when committing the changes—checking for conformance at each statement is the default. You can enable and disable constraints. Enable constraints with the NOVALIDATE clause to save time after large data loads.

References

[1] Oracle9i DBA Fundamentals I.