SECTION IV: Advance SQL

11. SQL PERFORMANCE TUNING

INDEXES

When a **SELECT** statement is fired to search for a particular record, the Oracle engine must first locate the table on the hard disk. The Oracle engine reads system information and finds the start location of a table's records on the current storage media. The Oracle engine then performs a sequential search to locate records that match user-defined criteria as specified in the **SELECT**.

For example, to locate all the accounts introduced by customer **C1** held in the **ACCT_MSTR** table, the Oracle engine must first locate the **ACCT_MSTR** table and then perform a table level, sequential search on the **INTRO_CUST_NO** column seeking a value equal to C1.

Records in the **ACCT_MSTR** table are stored in the order in which they are keyed. Thus to get all accounts where **INTRO_CUST_NO** is equal to C1 the Oracle engine must search the **entire** table column.

Indexing a table is an **access strategy**, that is, a way to sort and search records in the table. Indexes are essential to improve the speed with which record(s) can be located and retrieved from a table.

Note

An index is an ordered list of the contents of a column, (or a group of columns) of a table.

Indexing involves forming a two dimensional matrix completely independent of the table on which the index is being created. This two dimensional matrix will have a **single column**, which will hold sorted data, extracted from the table column(s) on which the index is created.

Another column called the **address field** identifies the location of the record in the Oracle database.

When data is inserted in the table, the Oracle engine automatically inserts the data value in the index. For every data value held in the index the Oracle engine inserts a unique **ROWID** value. This is done for every data value inserted into the index, without exception. This **ROWID** indicates exactly where the record is stored in the table.

Hence once the appropriate index data values have been located, the Oracle engine locates an associated record in the table using the **ROWID** found in the table.

The records in the index are sorted in the ascending order of the index column(s).

If the SELECT statement has a **WHERE** clause bound to a table column that is indexed, the Oracle engine will **scan the index** sequentially looking for a match of the search criteria **rather than** the table column itself. The sequential search is done using an ASCII compare routine to scan the columns of an index.

Since the data is sorted on the indexed column(s), the sequential search ends as soon as the Oracle engine reads an index data value that **does not meet** the search criteria.

Address Field In The Index

Each table in an Oracle database internally has a pseudocolumn named ROWID. This pseudocolumn is not evident when listing the structure of a table by executing a **SELECT** * statement, or a **DESCRIBE** statement using SQL*Plus, nor does the pseudocolumn take up space in the table. However, each row's address can be retrieved with a SQL query using the reserved word ROWID as a column name as:

SELECT *ROWID*, ACCT_NO **FROM** ACCT_MSTR;

Output:

```
ROWID              ACCT_NO
-------------------------
AAAHdpAABAAAMWCAAA SB1
AAAHdpAABAAAMWCAAB CA2
AAAHdpAABAAAMWCAAC SB3
AAAHdpAABAAAMWCAAD CA4
AAAHdpAABAAAMWCAAE SB5
AAAHdpAABAAAMWCAAF SB6
AAAHdpAABAAAMWCAAG CA7
AAAHdpAABAAAMWCAAH SB8
AAAHdpAABAAAMWCAAI SB9
AAAHdpAABAAAMWCAAJ CA10
10 rows selected.
```

The value of the pseudocolumn **ROWID** cannot be set or deleted using the **INSERT** or **UPDATE** statements. Oracle uses the **ROWID** values in the pseudocolumn ROWID internally for the construction of indexes. ROWIDs can be referenced like other table columns in **SELECT statements** and **WHERE clauses**, but cannot be stored in the database.

The address field of an index is called **ROWID**. **ROWID** is an internally generated and maintained value, which **uniquely** identifies a record. The information in the **ROWID** column provides the Oracle engine the location of the table and a specific record in the Oracle database.

Oracle uses ROWIDs internally for the construction of indexes. Each key in an index is associated with a ROWID that points to the associated row's address for fast access.

Users and application developers can also use ROWIDs for the following functions:

- Rowids are the fastest means of accessing particular rows
- Rowids can be used to see how a table is organized
- Rowids are unique identifiers for rows in a given table

A ROWID datatype are of two formats:

- **Extended:** The extended ROWID format supports tablespace-relative data block addresses and efficiently identifies rows in partitioned tables and indexes as well as nonpartitioned tables and indexes. Tables and indexes created by an Oracle8i or higher server always have extended ROWIDs
- **Restricted:** A restricted ROWID format is also available for backward compatibility with applications developed with Oracle7 or earlier releases

The **ROWID** format used by Oracle for **Restricted format** is as follows:

BBBBBBB.RRRR.FFFF

where, **FFFF** is a unique number given by the Oracle engine to each **Data File**. Data files are the files used by the Oracle engine to store user data.

For example, a database can be a collection of data files as follows:

Data File Name	Data File Number	Size of the Data Files
Sysorcl.ora	1	10 MB
Temporcl.ora	2	5 MB
Sctstaff.ora	3	30 MB
Sctstudent.ora	4	30 MB

Each data file is given a unique number at the time of data file creation. The Oracle engine uses this number to identify the data file in which, sets of table records are stored.

Each data file is further divided into **Data Blocks** and each block is given a unique number. The unique number assigned to the first data block in a data file is **0**. Thus block number can be used to identify the data block in which a record is stored. **BBBBBBB** is the block number in which the record is stored.

Each data block can store one or more **Records**. Thus each record in the data block is given a unique record number. The unique record number assigned to the first record in each data block is 0. Thus record number can be used to identify a record stored in a block. **RRRR** is a unique record number.

Each time a record is inserted into the table, Oracle locates free space in the **Data Blocks** in the data files. Oracle then inserts a record in the table and makes an entry in the index. The entry made in the index consists of table data combined with the Oracle engine created **ROWID** for the table record.

Thus, in a **restricted** format, data in an index will be represented as follows:

Data Field	Address Field
SB1	00000440.0000.0003
CA2	00000440.0001.0003
SB3	00000440.0002.0003
CA4	00000441.0000.0003
SB5	00000441.0001.0003

The **ROWID** format used by Oracle for **Extended format** is as follows:

OOOOOOFFFBBBBBBRRR

where,

OOOOOO is the data object number that identifies the database segment (i.e. **AAAHdp** in the example below). Schema objects in the same segment, such as a cluster of tables, will have the same data object number.

FFF is the TABLESPACE-relative datafile number of the datafile that contains the row (i.e. **AAB** in the example below).

BBBBBB is the data block that contains the row (i.e. **AAAMWC** in the example below). Block numbers are relative to their datafile, not tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.

RRR is the row in the block (i.e. **AAA** in the example below).

Thus, in an **extended** format, data in an index will be represented as follows:

Data Field	Address Field
SB1	AAAHdpAABAAAMWCAAA
CA2	AAAHdpAABAAAMWCAAB
SB3	AAAHdpAABAAAMWCAAC
CA4	AAAHdpAABAAAMWCAAD
SB5	AAAHdpAABAAAMWCAAE

To retrieve data from an Oracle table at the fastest possible speed, the Oracle engine requires a Search **Criteria** (i.e. the value to look for in the index).

Since the data in the index is sorted, the sequential search ends as soon as the Oracle engine reads an index data value that does not meet the search criteria. Thus, Oracle engine need not search the entire indexed column. This sharply reduces data retrieval time.

Once the data value in an index is located, the address field in an index specifies a **ROWID** which points to a data file, block and the record number directly. Thus the time taken by the Oracle engine to locate table data on the hard disk is reduced and data retrieval time is vastly improved.

Example 1:

Show all those account number along with the account opening date verified by the employee **E1**. There is no index on the field VERI_EMP_NO created for the ACCT_MSTR table.

Solution:

```
SELECT ACCT_NO, OPNDT, VERI_EMP_NO FROM ACCT_MSTR
    WHERE VERI_EMP_NO = 'E1';
```

Output:

```
ACCT_NO  OPNDT          VERI_EMP_NO
------------------------------------
SB1      05-NOV-03      E1
CA2      10-NOV-03      E1
SB5      15-DEC-03      E1
SB8      29-JAN-04      E1
SB11     10-MAR-04      E1
CA12     10-MAR-04      E1
SB15     15-APR-04      E1
```

Explanation:

When the above select statement is executed, since an index is not created on the **VERI_EMP_NO** column, the Oracle engine will scan the Oracle system information to locate the table in the data file. The Oracle engine will then perform a sequential search to retrieve records that match the search criteria (i.e. VERI_EMP_NO = E1) by comparing the value in the search criteria with the value in the **VERI_EMP_NO** column from the first record to the last record in the table.

Example 2:

Show all those account number along with the account opening date verified by the employee **E1**. There is an index on the field VERI_EMP_NO created for the ACCT_MSTR table.

Since an index exists on the **VERI_EMP_NO** column of the **ACCT_MSTR** table, the index data will be represented as follows:

Index Name: **idxVeriEmpNo**

VERI_EMP_NO	ROWID
E1	AAAHeeAABAAAMWCAAA
E4	AAAHeeAABAAAMWCAAC
E1	AAAHeeAABAAAMWCAAE
E4	AAAHeeAABAAAMWCAAG
E4	AAAHeeAABAAAMWCAAI
E1	AAAHeeAABAAAMWCAAK
E4	AAAHeeAABAAAMWCAAM
E1	AAAHeeAABAAAMWCAAO
E1	AAAHeeAABAAAMWCAAB
E4	AAAHeeAABAAAMWCAAD
E4	AAAHeeAABAAAMWCAAF
E1	AAAHeeAABAAAMWCAAH
E4	AAAHeeAABAAAMWCAAJ
E1	AAAHeeAABAAAMWCAAL
E4	AAAHeeAABAAAMWCAAN

Note

The index is in the **ascending** order of VERI_EMP_NO. The addresses have been assigned a data object number, a datafile number, a data block numbers and a row number in the order of **creation.**

Solution:

```
SELECT ACCT_NO, OPNDT, VERI_EMP_NO FROM ACCT_MSTR
    WHERE VERI_EMP_NO = 'E1';
```

When the above select statement is executed, since an index is created on **VERI_EMP_NO** column, the Oracle engine will scan the index to search for a specific data value (**i.e.** VERI_EMP_NO = E1). The Oracle engine will then perform a sequential search to retrieve records that match the search criteria (i.e. VERI_EMP_NO = E1). When **E2** is read, the Oracle engine stops further retrieval from the index.

For the seven records retrieved, the Oracle engine locates the address of the table records from the **ROWID** field and retrieves records stored at the specified address.

Output:

```
VERI_EMP_NO  ROWID
E1           AAAHeeAABAAAMWCAAA
E1           AAAHeeAABAAAMWCAAB
E1           AAAHeeAABAAAMWCAAE
E1           AAAHeeAABAAAMWCAAH
E1           AAAHeeAABAAAMWCAAK
E1           AAAHeeAABAAAMWCAAL
E1           AAAHeeAABAAAMWCAAO
```

The **Rowid** in the current example indicates that the records with VERI_EMP_NO E1 are located in **data object** numbered **AAAHee** having **datafile** numbered **AAB, data block** numbered **AAAMW** and the rows number as **AA_**.

Thus data retrieval from a table by using an index is much faster than data retrieval from the table where indexes are not defined.

Duplicate / Unique Index

Oracle allows the creation of two types of indexes. These are:

- ❑ Indexes that **allow** duplicate values for the indexed columns **i.e. Duplicate Index**
- ❑ Indexes that **deny** duplicate values for the indexed columns **i.e. Unique Index**

Creation Of An Index

An index can be created on one or more columns. Based on the number of columns included in the index, an index can be:

- ❑ Simple Index
- ❑ Composite Index

Creating Simple Index

An index created on a single column of a table is called a **Simple Index**. The syntax for creating simple index that allows duplicate values is as described.

Syntax:

```
CREATE INDEX <IndexName> ON <TableName> (<ColumnName>);
```

Example 3:

Create a simple index on **VERI_EMP_NO** column of the **ACCT_MSTR** table.

Solution:

```
CREATE INDEX idxVeriEmpNo ON ACCT_MSTR (VERI_EMP_NO);
```

Output:

```
Index created.
```

Creating Composite Index

An index created on more than one column is called a **Composite Index**. The syntax for creating a composite index that allows duplicate values is:

Syntax:

```
CREATE INDEX <IndexName>
    ON <TableName> (<ColumnName1>, <ColumnName2>);
```

Example 4:

Create a composite index on the **TRANS_MSTR** table on columns **TRANS_NO** and **ACCT_NO**

Solution:

```
CREATE INDEX idxTransAcctNo ON TRANS_MSTR (TRANS_NO, ACCT_NO);
```

Output:

```
Index created.
```

Note

The indexes in the above examples do not enforce uniqueness **i.e.** the columns included in the index can hold duplicate values. To create unique index, the keyword **UNIQUE** should be included in the **Create Index** command.

Creation of Unique Index

A unique index can also be created on one or more columns. If an index is created on a single column, it is called a **Simple Unique Index**. The syntax for creating a simple unique index is as follows:

Syntax:

```
CREATE UNIQUE INDEX <IndexName> ON <TableName> (<ColumnName>);
```

If an index is created on more than one column, it is called a **Composite Unique Index**. The syntax for creating a composite unique index is as follows:

Syntax:

```
CREATE UNIQUE  INDEX <IndexName>
  ON <TableName> (<ColumnName>, <ColumnName>);
```

Example 5:

Create a unique index on **CUST_NO** column of the **CUST_MSTR** table.

Solution:

CREATE UNIQUE INDEX idx_CustNo **ON** CUST_MSTR **(**CUST_NO**);**

Output:

```
Index created.
```

Note

When the user defines a primary key or a unique key constraint at table or column level, the Oracle engine automatically creates a **unique index** on the primary key or unique key column(s).

Reverse Key Indexes

Creating a reverse key index, when compared to a simple index, reverses each byte of the column being indexed while keeping the column order. Such an arrangement can help avoid performance degradation in indexes where modifications to the index are concentrated on a small set of blocks. By reversing the keys of the index, the insertions become distributed all over the index.

For example, the column value is stored in an index as shown below:

In normal index	In reverse index
C1	1C
C2	2C
C3	3C
C4	4C

Here, column values are stored in a normal index. Then the rows will be stored together in one block as the values are almost the same for all rows. When the same column is indexed in reverse mode then the column values will be stored in different blocks as the starting value differs.

Using the key arrangement eliminates the ability to run an index range-scanning query on the index. As lexically adjacent keys are not stored next to each other in a reverse key index, only fetch-by-key or full-index (table) scans can be performed.

Under some circumstances, using a reverse-key index can make an application run faster.

Syntax:

```
CREATE INDEX <IndexName>
    ON <TableName> (<ColumnName>) REVERSE
```

Example 6:

Create a reverse index on **CUST_NO** column of the **CUST_MSTR** table.

Solution:

CREATE INDEX idx_CustNo **ON** CUST_MSTR (CUST_NO) **REVERSE;**

Output:

```
Index created.
```

A reverse key index can be rebuilt into a normal index using the keywords **REBUILD NOREVERSE**.

Syntax:

```
ALTER INDEX <IndexName> REBUILD NOREVERSE;
```

Example 7:

Modify the reverse index just created to a normal index on **CUST_NO** column of the **CUST_MSTR** table.

```
ALTER INDEX idx_CustNo REBUILD NOREVERSE;
```

Note

A normal index **cannot** be rebuilt as a reverse key index.

Bitmap Indexes

The advantages of using bitmap indexes are greatest for low cardinality columns **i.e.** columns in which the number of distinct values is small compared to the number of rows in the table. If the values in a column are repeated more than a hundred times, the column is a candidate for a bitmap index. For example, in a table with one million rows, rows with 10,000 distinct values are candidates for a bitmap index.

Syntax:

```
CREATE BITMAP INDEX <IndexName> ON <TableName> (<ColumnName>);
```

Example 8:

Create a bitmap index on **TRANS_NO** column of the **TRANS_MSTR** table.

```
CREATE BITMAP INDEX bitidx_TransNo ON TRANS_DTLS (TRANS_NO);
```

Output:

```
Index created.
```

Bitmap indexing provides the following benefits:

- Reduced response time for large classes of ad hoc queries
- A substantial reduction of space usage compared to other indexing techniques
- Dramatic performance gains even on very low end hardware

Fully indexing a large table with a normal index can be prohibitively expensive in terms of space since the index can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

In adhoc queries or similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the **WHERE** clause of a query can be quickly resolved by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to rowids.

If the resulting number of rows is small, the query can be answered very quickly without resorting to a full scan of the table.

Function Based Index

A column's index **will not** be used when the same column is expressed in an arithmetic expression or function in the **WHERE** clause.

To facilitate such an operation Oracle **allows** creating indexes based on a function or expression mapped to one or more columns in a table. Function based indexes are very useful when the **where clause** contains functions or expressions to evaluate a query.

The function used for building the index can be an arithmetic expression or an expression that contains a PL/SQL function, package function, or SQL function. The expression **cannot contain** any aggregate functions. A function-based index cannot be created on a LOB column, REF, nested table column or the object type contains a LOB, REF, or nested table.

Syntax:

```
CREATE INDEX <IndexName> ON <TableName> (<Function>(<ColumnName>));
```

Example 9:
Create an index on the function **UPPER** used on **FNAME** column of the **CUST_MSTR** table.

```
CREATE INDEX idx_Name ON CUST_MSTR (UPPER(FNAME));
```

Output:

```
Index created.
```

Key-Compressed Index

Key compression breaks an index key into a prefix and a suffix entry. Compression is achieved by sharing the prefix entries among all the suffix entries in an index block. This sharing can lead to huge savings in space, allowing more keys to be stored per index block.

Key compression can be useful when in a non-unique index the ROWID is appended to make the key unique. When key compression is used, the duplicate key will be stored as a prefix entry on the index block without the ROWID. The remaining rows will be suffix entries consisting of only the ROWID.

Syntax:

```
CREATE INDEX <IndexName> ON <TableName>
    (<ColumnName1>, <ColumnName2>, ...) COMPRESS 1
```

For unique indexes, the valid range of prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.

For non-unique indexes, the valid range of prefix length values is from 1 to the number of key columns. The default prefix length is the number of key columns.

Dropping Indexes

Indexes associated with the tables can be removed by using the **DROP INDEX** command.

Syntax:

```
DROP INDEX <indexname>;
```

Example 10:
Remove index **idx_CustNo** created for the table **CUST_MSTR.**

```
DROP INDEX idx_CustNo;
```

Note

When a table, which has associated indexes (unique or non-unique), is dropped, the Oracle engine automatically drops all the associated indexes as well.

MULTIPLE INDEXES ON A TABLE

The Oracle engine allows creation of multiple indexes on each table. The Oracle engine prepares a query **plan** to decide on the index that must be used for specific data retrieval based on the **WHERE clause** or the **ORDER BY clause** specified in the SELECT statement.

Whenever a SELECT statement is executed, the Oracle engine prepares a query plan that identifies the data retrieval method. The query plan (among other information) holds the name of the table from which data will be retrieved and the name of the index that must be used for data retrieval.

> **Note** If a SELECT statement is fired without a where clause and without an order by clause the Oracle engine **does not** use the indexes created on the table for data extraction.

- If a where clause or an ORDER BY clause is specified, the Oracle engine uses the index created on a column on which the where clause or the order by clause is specified
- If there is no index for the column specified in the WHERE clause or the ORDER BY clause is not created, data is retrieved without using indexes

Instances When The Oracle Engine Uses An Index For Data Extraction

- A SELECT statement with WHERE clause specified on the column on which an index exists
- A SELECT statement with ORDER BY clause specified on the column on which an index exists

Instances When The Oracle Engine Does Not Use An Index For Data Extraction

- A SELECT statement without search criteria and order by clause
- A SELECT statement with WHERE clause specified on the column on which an index is not defined
- A SELECT statement with ORDER BY clause specified on the column on which an index is not defined

Too Many Indexes - A Problem

Each time a record is inserted into the table:

- The Oracle engine locates free space in the blocks in the data files
- Then inserts a record in all the indexes associated with the table
- The index entries are sorted in the ascending order as well

If too many indexes are created on a table the Oracle engine will take longer to insert a record in a table since index processing must be done for each record that is inserted, updated or deleted.

Thus while indexes speeds up data retrieval, data insertion slows down considerably. A balance must be maintained such that only columns that are frequently used for data retrieval (i.e. querying the table) are indexed.

USING ROWID TO DELETE DUPLICATE ROWS FROM A TABLE

Retaining one row in table, while deleting all other duplicate rows, is quite an interesting exercise. If the delete statement contains a WHERE clause based on **EMP_NO** then **all rows** in the table will be deleted immediately.

For example, if the data in the **EMP_MSTR** table is:

EMP_NO	FNAME	DEPT
E1	*Ivan*	*Administration*
E1	*Ivan*	*Administration*
E1	*Ivan*	*Administration*
E2	Amit	Loans And Financing
E3	Maya	Client Servicing

EMP_NO	FNAME	DEPT
E4	Peter	Loans And Financing
E5	Mandhar	Marketing
E6	Sonal	Administration
E7	Anil	Marketing
E8	Seema	Client Servicing
E9	Vikram	Marketing
E10	Anjali	Administration

And a delete statement is executed as:

DELETE FROM EMP_Master **WHERE** EMP_NO **IN**('E1' , 'E2', 'E3');

All records with EMP_NO E1, E2 or E3 will be deleted immediately.

However, what is required is that the Oracle engine **must retain one record** and delete all other duplicate records. To retain one record, the where clause must be defined on the column that **uniquely** identifies each record.

As seen earlier, even if user enters duplicate records, the Oracle engine will assign a unique **rowid** value that points to a record within a block, in the data file, for each record entered by the user.

A specific record in a table will be stored within a block in the data file. Each record in a block is given a unique record number. Thus at any time the value in the rowid column will always be unique.

A DELETE statement must be written such that the WHERE clause is defined using the **rowid** column. The values for the WHERE clause in the DELETE statement must be identified by using a SELECT statement that retrieves the rowid of the first row in each set of duplicate records in the table.

Then when a WHERE clause is specified in the DELETE statement with the **NOT IN** operator it deletes all duplicate rows but isolates one row in each set.

A **subquery** is an SQL statement that extracts values from table columns using a SELECT statement and passes these values as input to another SQL statement. The SELECT statement is called **Inner SQL statement** and the SQL statement to which the values of the select statement are passed is called **Parent SQL statement**. The parent SQL statement can be an INSERT, UPDATE, DELETE, SELECT or CREATE TABLE statement.

The Oracle engine executes the inner SELECT statement and then processes the parent SQL statement based on the values retrieved by the inner SELECT statement.

Inner Select Statement

To create a record set of identical records from a table, the records must be grouped on all the columns in the table by using a GROUP BY clause in the SELECT statement.

A SELECT statement will then retrieve the ROWID of the first row in each set of duplicate records. The first row in each set can be extracted by using the **MIN** function that returns the minimum value from a set of values. Thus the select statement will be:

SELECT MIN(ROWID) FROM EMP_MSTR **GROUP BY** EMP_NO, FNAME, DEPT ....

Parent SQL Statement

In the current example the Parent SQL statement will be a DELETE statement that will delete the records based on the ROWID fetched by the Inner SQL statement.

The query used to delete duplicate rows will be:

Example 11:

```
DELETE FROM EMP_MSTR WHERE ROWID NOT IN(SELECT MIN(ROWID)
    FROM EMP_MSTR GROUP BY EMP_NO, FNAME, DEPT);
```

When the inner SELECT statement is executed, data is grouped on all the columns of the table and the **MIN** function returns the minimum ROWID in the group. Thus the output held in memory will be as follows:

```
ROWID               EMP_NO  FNAME    DEPT
------------------------------------------------------------------
AAAHebAABAAAMVqAAA  E1      Ivan     Administration
AAAHebAABAAAMVqAAB  E2      Amit     Loans And Financing
AAAHebAABAAAMVqAAC  E3      Maya     Client Servicing
AAAHebAABAAAMVqAAD  E4      Peter    Loans And Financing
AAAHebAABAAAMVqAAE  E5      Mandhar  Marketing
AAAHebAABAAAMVqAAF  E6      Sonal    Administration
AAAHebAABAAAMVqAAG  E7      Anil     Marketing
AAAHebAABAAAMVqAAH  E8      Seema    Client Servicing
AAAHebAABAAAMVqAAI  E9      Vikram   Marketing
AAAHebAABAAAMVqAAJ  E10     Anjali   Administration
```

The Oracle engine after the execution of the inner SELECT statement replaces the SELECT statement with the minimum ROWID for each group as retrieved by the SELECT statement. Thus the delete statement will be changed to:

```
DELETE FROM EMP_MSTR WHERE ROWID NOT IN('AAAHebAABAAAMVqAAA',
    'AAAHebAABAAAMVqAAB', 'AAAHebAABAAAMVqAAC', 'AAAHebAABAAAMVqAAD',
    'AAAHebAABAAAMVqAAE', 'AAAHebAABAAAMVqAAF', 'AAAHebAABAAAMVqAAG',
    'AAAHebAABAAAMVqAAH', 'AAAHebAABAAAMVqAAI', 'AAAHebAABAAAMVqAAJ');
```

Thus all records with rowid other than those in the list specified above are deleted.

If a select statement is executed on the **EMP_MSTR** table after such a delete operation, the Oracle engine displays the following output for the query.

```
SELECT EMP_NO, FNAME, DEPT FROM EMP_MSTR;
```

Output:

```
EMP_NO  FNAME    DEPT
---------------------------------------------
E1      Ivan     Administration
E2      Amit     Loans And Financing
E3      Maya     Client Servicing
E4      Peter    Loans And Financing
E5      Mandhar  Marketing
E6      Sonal    Administration
E7      Anil     Marketing
E8      Seema    Client Servicing
E9      Vikram   Marketing
E10     Anjali   Administration
```

Using this technique, duplicate records can be deleted from the table while maintaining one record in the table for reference.

USING ROWNUM IN SQL STATEMENTS

For each row returned by a query, the **ROWNUM** pseudo column returns a number indicating the order in which Oracle engine selects the row from a table or set of joined rows. The first row selected has a **ROWNUM** of **1**, the second has **2**, and so on.

Using ROWNUM To Limit Number Of Rows In A Query

ROWNUM can be used to limit the number of rows retrieved.

Example 12:
Retrieve first three rows from the BRANCH_MSTR table using ROWNUM
Table Name: BRANCH_MSTR

BRANCH_NO	NAME
B1	Vile Parle (HO)
B3	Churchgate
B5	Borivali

BRANCH_NO	NAME
B2	Andheri
B4	Mahim
B6	Darya Ganj

SELECT ROWNUM, BRANCH_NO, NAME **FROM** BRANCH_MSTR **WHERE ROWNUM** < 4;

Output:

```
ROWNUM BRANCH_NO   NAME
------------------------------------------
     1 B1          Vile Parle (HO)
     2 B2          Andheri
     3 B3          Churchgate
```

Caution

The Oracle engine assigns a **ROWNUM** value to each row as it is retrieved, before rows are sorted on the column(s) in the ORDER BY clause. The order in which data is retrieved is dependent upon the indexes created on the table.

If an index is created on the column(s) used in the order by clause, the Oracle engine uses the index to retrieve data in a sorted order. Thus the ROWNUM will be in the order of the rows retrieved from the index.

If an index is not created on the column(s) used in the order by clause, the Oracle engine will retrieve data from the table in the order of data insertion and thus an ORDER BY clause does not affect the ROWNUM of each row.

VIEWS

After a table is created and populated with data, it may become necessary to prevent all users from accessing all columns of a table, for data security reasons. This would mean creating several tables having the appropriate number of columns and assigning specific users to each table, as required. This will answer data security requirements very well but will give rise to a great deal of redundant data being resident in tables, in the database.

To reduce **redundant data** to the minimum possible, Oracle allows the creation of an object called a **View**. A View is mapped, to a SELECT sentence. The table on which the view is based is described in the FROM clause of the SELECT statement. The SELECT clause consists of a sub-set of the columns of the table. Thus a View, which is mapped to a table, will in effect have a sub-set of the actual columns of the table from which it is built. This technique offers a simple, effective way of hiding columns of a table.

An interesting fact about a View is that it is stored only as a definition in Oracle's system catalog. When a reference is made to a View, its definition is scanned, the base table is opened and the View created on top of the base table. Hence, a view holds no data at all, until a specific call to the view is made. This reduces redundant data on the HDD to a very large extent. When a View is used to manipulate table data, the underlying base table will be completely invisible. This will give the level of data security required.

The Oracle engine treats a View just as though it was a base table. Hence a View can be queried exactly as though it was a base table. However, a query fired on a view will run slower than a query fired on a base table. This is because the View definition has to be retrieved from Oracle's system catalog, the base table has to be identified and opened in memory and then the View has to be constructed on top of the base table, suitably masking table columns. Only then will the query actually execute and return the active data set.

Some Views are used only for looking at table data. Other Views can be used to Insert, Update and Delete table data as well as View data. If a View is used to only look at table data and nothing else the View is called a **Read-Only** View. A View that is used to look at table data as well as Insert, Update and Delete table data is called an **Updateable** View.

The reasons why views are created are:

- When Data security is required
- When Data redundancy is to be kept to the minimum while maintaining data security

Lets spend some time in learning how a View is:

- Created
- Used for only viewing and / or manipulating table data (**i.e.** a read-only or updateable view)
- Destroyed

Creating View

Syntax:

```
CREATE VIEW <ViewName> AS
    SELECT <ColumnName1>, <ColumnName2> FROM <TableName>
        WHERE <ColumnName> = <Expression List>;
        GROUP BY <Grouping Criteria> HAVING <Predicate>
```

Note

The ORDER BY clause **cannot** be used while creating a view.

Example 13:

Create a view called Customers on the CUST_MSTR table.

CREATE VIEW vw_Customers **AS SELECT * FROM** CUST_MSTR;

Note

The columns of the table are related to the view using a **one-to-one** relationship.

Example 14:

Create a view called Employees on the **EMP_MSTR** table.

CREATE VIEW vw_Employees **AS** SELECT FNAME, MNAME, LNAME, DEPT
  FROM EMP_MSTR;

This creates a view by the name of **vw_Employees** based on the table **EMP_MSTR**.

Renaming The Columns Of A View

The columns of the view can take on different names from the table columns, if required.

Example 15:

```
CREATE VIEW vw_Transactions AS
    SELECT ACCT_NO "Account No.", DT "Date", Type, DR_CR "Mode", AMT "Amount"
        FROM TRANS_MSTR;
```

Selecting A Data Set From A View

Once a view has been created, it can be queried exactly like a base table.

Syntax:

```
SELECT <ColumnName1>,<ColumnName2> FROM <ViewName>;
```

Note

Instead of a **table** name in the FROM clause, a **view** name is used. The SELECT statement can have all the clauses like WHERE, ORDER BY etc.

Example 15:

```
SELECT FNAME, LNAME, DEPT FROM vw_Employees
    WHERE DEPT IN('Marketing', 'Loans And Financing');
```

Updateable Views

Views can also be used for data manipulation (**i.e.** the user can perform the Insert, Update and Delete operations). Views on which data manipulation can be done are called **Updateable Views**. When an **updateable view name** is given in an Insert Update, or Delete SQL statement, modifications to data in the view will be immediately passed to the underlying table.

For a view to be updateable, it should meet the following criteria:

- ❑ Views defined from Single table
- ❑ If the user wants to **INSERT** records with the help of a view, then the PRIMARY KEY column(s) and **all** the NOT NULL columns must be included in the view
- ❑ The user can **UPDATE, DELETE** records with the help of a view even if the PRIMARY KEY column and NOT NULL column(s) **are excluded** from the view definition

Example 16:

Table Name: NOMINEE_MSTR

Column Name	Data Type	Width	Attributes
NOMINEE_NO	VarChar2	10	Primary key
ACCT_FD_NO	VarChar2	10	Not Null
NAME	VarChar2	75	Not Null
DOB	Date		
RELATIONSHIP	VarChar2	25	

```
CREATE VIEW vw_Nominees AS
  SELECT NOMINEE_NO, ACCT_FD_NO, NAME FROM NOMINEE_MSTR;
```

When an INSERT operation is performed using the view:

```
INSERT INTO vw_Nominees VALUES('N100', 'SB432', 'Sharanam');
```

Oracle returns the following message:

```
1 row created
```

When a MODIFY operation is performed using the view:

```
UPDATE vw_Nominees SET NAME = 'Vaishali' WHERE NAME='Sharanam';
```

Oracle returns the following message:

```
1 row updated.
```

When a DELETE operation is performed using the view

```
DELETE FROM vw_Nominees WHERE NAME = 'Vaishali';
```

Oracle returns the following message:

```
1 row deleted.
```

A view can be created from more than one table. For the purpose of creating the View these tables will be linked by a **join** specified in the **WHERE clause** of the View definition.

The behavior of the View will vary for Insert, Update, Delete and Select table operations depending upon the following:

- Whether the tables were created using a Referencing clause
- Whether the tables were created without any Referencing clause and are actually standalone tables not related in any way

Views Defined From Multiple Tables (Which Have No Referencing Clause)

If a view is created from multiple tables, which were not created using a **Referencing clause (i.e.** No logical linkage exists between the tables), then though the PRIMARY Key Column(s) as well as the NOT NULL columns are included in the View definition the view's behavior will be as follows:

The INSERT, UPDATE or DELETE operation is **not allowed**. If attempted, Oracle displays the following error message:

For insert/modify:

```
ORA -01779: cannot modify a column, which maps to a non key-preserved
table.
```

For delete:

```
ORA -01752: cannot delete from view without exactly one key-preserved
table.
```

Views Defined From Multiple Tables (Which Have Been Created With A Referencing Clause)

If a view is created from multiple tables, which were created using a **Referencing clause** (i.e. a logical linkage exists between the tables), then though the **PRIMARY Key** Column(s) as well as the **NOT NULL** columns are included in the View definition, the view's behavior will be as follows:

- An **INSERT** operation is not allowed
- The **DELETE** or **MODIFY** operations **do not** affect the Master table
- The view can be used to **MODIFY** the columns of the **detail table** included in the view

If a **DELETE** operation is executed on the view, the corresponding records from the **detail table** will be deleted.

Example 17:

Table Name: BRANCH_MSTR

Column Name	Data Type	Size	Attributes
BRANCH_NO	VarChar2	10	Primary Key / First letter must be 'B'
NAME	VarChar2	25	

Table Name: ADDR_DTLS

Column Name	Data Type	Size	Attributes
ADDR_NO	Number	6	Primary Key
CODE_NO	VarChar2	10	Foreign Key references BRANCH_NO of the BRANCH_MSTR table.
ADDR_TYPE	VarChar2	1	Can hold the values: **H** for Head Office or **B** for Branch
ADDR1	VarChar2	50	
ADDR2	VarChar2	50	
CITY	VarChar2	25	
STATE	VarChar2	25	
PINCODE	VarChar2	6	

Syntax for creating a Master/Detail View

```
CREATE VIEW vw_Branch AS
    SELECT BRANCH_NO, NAME, ADDR_TYPE, ADDR1, ADDR2, CITY, STATE, PINCODE
        FROM BRANCH_MSTR, ADDR_DTS
        WHERE ADDR_DTLS.CODE_NO = BRANCH_MSTR.BRANCH_NO;
```

When an INSERT operation is performed using the view

```
INSERT INTO vw_Branch VALUES('B7', 'Dahisar', 'B', 'Vertex Plaza, Shop 4,', 'Western Express Highway,
                                Dahisar (East),', 'Mumbai', 'Maharashtra', '400078');
```

Oracle returns the following error message:

```
ORA-01776: cannot modify more than one base table through a join view
```

When a MODIFY operation is performed using the view

```
UPDATE vw_Branch SET PINCODE = '400079' WHERE BRANCH_NO = 'B5';
```

Oracle returns the following message:

```
1 row updated.
```

When a DELETE operation is performed using the view

```
SQL> DELETE FROM vw_Branch WHERE BRANCH_NO = 'B5';
```

Oracle returns the following message:

```
1 row deleted.
```

Common Restrictions On Updateable Views

The following condition holds true irrespective of the view being created from a single table or multiple tables.

For the view to be updateable the view definition must not include:

- Aggregate functions
- DISTINCT, GROUP BY or HAVING clause
- Sub-queries
- Constants, Strings or Value Expressions like Sell_price * 1.05
- UNION, INTERSECT or MINUS clause
- If a view is defined from another view, the second view should be updateable

If the user tries to perform any of INSERT, UPDATE, DELETE operation, on a view, which is created from a non-updateable view Oracle returns the following error message

FOR INSERT/MODIFY/DELETE

```
ORA-01732: data manipulation operation not legal on this view
```

Destroying A View

The **DROP VIEW** command is used to remove a view from the database.

Syntax:

```
DROP VIEW <ViewName>;
```

Example 18:

Remove the view **vw_Branch** from the database.

DROP VIEW vw_Branch;

CLUSTERS

Clustering is an important concept for improving Oracle performance. Whenever the database is accessed, any reduction in input / output (**i.e.** I/O) always helps in improving it's throughput and overall performance. The concept of a **cluster** is where member records are stored **physically** near parent records. For Oracle, clusters can be used to define common, one-to-many access paths, and the member rows can be stored on the same database block as their owner row.

Clusters are used to store data from different tables in the **same physical data blocks**. They are appropriate to use if the records from those tables are frequently queried together. By storing them in the same data blocks, the number of database block **reads** needed to fulfill such queries decreases, thereby improving performance.

Note

Clusters may have a negative performance impact on the data manipulation transactions and on queries that only reference one of the tables in the cluster.

Because of their unique structure, clustered tables have different storage requirements from non-clustered tables. Each cluster stores the table's data, as well as maintains the cluster index that is used to sort table data.

The Cluster Key

The columns within the cluster index are called the **cluster key** (**i.e.** the set of columns that the tables in the cluster have in common). Since the cluster key columns determine the physical placement of rows within the cluster, the cluster key is usually the foreign key of one table that references the primary key of another table in the cluster.

After the cluster has been created, the cluster index is created on the cluster key columns. After the cluster key index has been created, data can be entered into the tables stored in the cluster. As rows are inserted, the database will store a cluster key and its associated rows in each of the cluster's blocks.

Syntax:

```
CREATE CLUSTER <ClusterName> (<Column> <DataType>
    [, <Column> <DataType>] . . .) [<Other Options>];
```

The cluster name follows the table naming conventions, also column and datatype is the name and datatype used as the cluster key. The column name may be same as one of the columns of a table or it may be any other valid name.

Example 19:

```
CREATE CLUSTER "DBA_BANKSYS"."BRANCH_INFO"("BRANCH_NO" VARCHAR2(10));

CREATE TABLE "DBA_BANKSYS"."BRANCH_MSTR"(
    "BRANCH_NO" VARCHAR2(10) PRIMARY KEY, "NAME" VARCHAR2(25))
    CLUSTER BRANCH_INFO(BRANCH_NO);

CREATE TABLE "DBA_BANKSYS"."ADDR_DTLS"(
    "ADDR_NO" NUMBER(6) PRIMARY KEY, "CODE_NO" VARCHAR2(10),
    "ADDR_TYPE" VARCHAR2(1), "ADDR1" VARCHAR2(50),
    "ADDR2" VARCHAR2(50), "CITY" VARCHAR2(25),
    "STATE" VARCHAR2(25), "PINCODE" VARCHAR2(6));
    CLUSTER BRANCH_INFO(BRANCH_NO);
```

Following are the advantages of Clusters:

- ❑ Disk I/O is reduced and access time improves for joins of clustered tables.
- ❑ In a cluster, a cluster key value is the value of the cluster key columns for a particular row. Each cluster key value is stored only once each in the cluster and the cluster index, no matter how many rows of different tables contain the value.
- ❑ Since all rows in clustered tables use the same columns as the common primary key, the columns are stored only once for all tables, yielding some storage benefit.

Following are the disadvantages of Clusters:

- ❑ Clusters can reduce the performance of INSERT statements as compared with storing a table separately with its own index.
- ❑ Columns that are updated often are not good candidates for the cluster key.

CLUSTER INDEXES

The Oracle Db engine normally handles huge amounts of user data within its tables. As table data goes on rapidly increasing, such as in transaction tables, the time taken for the Oracle DB engine to retrieve data from these tables also increases. Hence, query execution time does get adversely impacted when referencing tables carrying GBs or TBs of data.

Oracle offers several techniques to help contain this problem and deliver acceptable query execution times. Storing user data in clusters is one of them. Clustering data on a hard disk plays an important role in reducing query execution time in very large databases.

A cluster is a group of one of more tables that have common columns, which are physically stored and accessed together, on the hard disk.

A table cluster is a group of tables whose data is stored within same data blocks in the Oracle database, since they share common columns and are often used together. When clustered tables are created, Oracle physically stores all rows for each table within the same data blocks on the hard disk. The cluster key value is a specific value within the cluster key column(s) bound to a specific data row.

For example, Order and OrderDetails tables have OrderNo as a common column. For a single row of data in the in Order table identified by OrderNo, there will be multiple rows of data linked by the same OrderNo in the OrderDetails table.

OrderDetails is used to store and manipulate all order details associated with a single order within the Orders table. Hence, both these tables are always used together.

When such tables are being created their table data can be clustered on the hard disk. Clustering forms a group of these two tables with the Oracle Db engine recognizing the group as special, i.e. they share a common column OrderNo.

Since a cluster stores the data in the same data blocks, the number of database block reads needed to fulfill such queries decreases, thus considerably improving query execution time.

Clustering such table data should be the choice since data records are queried together and frequently from both tables.

Creating table clusters does not impact commercial application design. The speed of application execution will not be adversely impacted even if clustering is done after its deployment. Application tables that are part of a cluster are transparent to users and to the application. Clustered tables are visible only to the Oracle Db engine and SQL syntax used to manipulate such table data.

Each such table cluster holds user data along with a **cluster index**, which is used to sort table data on demand.

WHEN TO CLUSTER

Here are some simple guidelines to help decide when to cluster tables:

□ Tables that are accessed frequently by an application, using complex join statements

Caution

Do not cluster tables if:

Tables are occasionally accessed using a join

If the common column, data values are frequently modified

Oracle takes longer to modify cluster key, column data, than normal index values in non clustered tables.

□ Tables that share a master detail relationship such as Orders and OrderDetails, Employees and Departments, Customer and Contact details and so on

Usually when retrieving data from master/detail tables, master records are retrieved along with their corresponding detail records.

If such tables are clustered, and detail records are stored in the same data block(s) as the master record, then when the master record is retrieved its associated detail records will be retrieved in the same read operation. This requires Oracle to perform a lot less I/O.

Caution

Do not cluster tables that are normally read globally [i.e. all rows, all columns].

A full table scan of a clustered table takes longer than a full table scan of a non clustered table. Oracle is forced to read a lot more data blocks because all the table data is stored together.

Tip

Clustering improves the performance of queries that select multiple detail records of the single master record, but does not adversely effect the performance of a full table scan on the master table.

TYPES OF CLUSTERS

A cluster can be either an **Indexed Cluster** or a **Hash Cluster**.

Indexed Clusters

In an indexed cluster, the Oracle Db engine stores rows having the same cluster key value together. Each distinct cluster key value is stored only once within a data block, regardless of the number of tables and rows in which the cluster key value occurs. This saves a considerable amount of disk space, especially when working with very large databases and significantly improves data retrieval performance.

After such an indexed cluster is created within data blocks on the hard disk, an index needs to be created on the cluster key value before any **Data Manipulation Language** [DML] statements can be issued against a table within the cluster. This index is called the **Cluster Index.**

A cluster index provides quick access to data rows within a cluster, based on the cluster key. If an SQL query is fired to locate a row within a cluster, based on a cluster key value, the Oracle Db engine searches the cluster index for the cluster key, value match. Then the Oracle Db engine locates the data row from within the cluster based on its RowID obtained from the cluster index. A RowID is always directly bound to a data block ID on the hard disk.

Hash Clusters

Oracle stores rows that have the same hash key value together, in a hash cluster. The hash value for a row is a value returned by the cluster's hash function.
To use hashing, a hash cluster is created and data tables are loaded into it. The Oracle Db engine physically stores the rows of a table in a hash cluster and retrieves them according to the results of a hash function.

Tip

Hash values are not actually stored in the cluster, although cluster key values are stored for every row in the cluster.

A hash cluster can be created by using Oracle's internal hash function.

Oracle uses a hash function to generate a distribution of numeric values called hash values that are based on specific cluster key values.

The key of a hash cluster, can be held within a single column or if necessary across multiple columns (i.e. a composite key).

To locate or store a row in a hash cluster, the Oracle DB engine applies the hash function to the cluster key value of the row. The resulting hash value returned by the hash function corresponds to a data block ID on the hard disk. The Oracle DB engine then uses this to read or write to the hard disk depending on the type of SQL statement being executed.

This type of data location process normally results in lot less hard disk I/O then when data is located in an indexed cluster because an index search is not required.

Caution

A cluster index cannot be created on a hash cluster. There is no reason to create an index on a hash cluster key.

When dealing with SQL queries that have equality operators applied on the cluster key value, Hashing offers excellent query execution times, when compared to normal index searches.

The cluster key in the equality condition is hashed and the corresponding hash key is usually found within a single read operation. In comparison, when seeking data from within an indexed table, the index key value must first be found within the index [usually requires several read operations] and then data row is located and read from the table [requires another read operation].

SEQUENCES

The quickest way to retrieve data from a table is to have a column in the table whose data uniquely identifies a row. By using this column and a specific value, in the **WHERE** condition of a **SELECT** sentence the Oracle engine will be able to identify and retrieve the row the fastest.

To achieve this, a constraint is attached to a specific column in the table that ensures that the column is never left empty and that the data values in the column are unique. Since human beings do data entry, it is quite likely that a duplicate value could be entered, which violates this constraint and the entire row is rejected.

If the value entered into this column is **computer generated** it will always fulfill the unique constraint and the row will always be accepted for storage.

Oracle provides an object called a **Sequence** that can generate numeric values. The value generated can have a maximum of 38 digits. A sequence can be defined to:

- ❑ Generate numbers in ascending or descending order
- ❑ Provide intervals between numbers
- ❑ Caching of sequence numbers in memory to speed up their availability

A sequence is an independent object and can be used with any table that requires its output.

Creating Sequences

The minimum information required for generating numbers using a sequence is:

- ❑ The starting number
- ❑ The maximum number that can be generated by a sequence
- ❑ The increment value for generating the next number.

This information is provided to Oracle at the time of sequence creation.

Syntax:

```
CREATE SEQUENCE <SequenceName>
    [INCREMENT BY <IntegerValue>
    START WITH <IntegerValue>
    MAXVALUE <IntegerValue> / NOMAXVALUE
    MINVALUE <integervalue>  / NOMINVALUE
    CYCLE / NOCYCLE
    CACHE <IntegerValue> / NOCACHE
    ORDER  / NOORDER]
```

Note

Sequence is always given a name so that it can be referenced later when required.

Keywords And Parameters

INCREMENT BY: Specifies the interval between sequence numbers. It can be any positive or negative value but not zero. If this clause is omitted, the default value is 1.

MINVALUE: Specifies the sequence minimum value.

NOMINVALUE: Specifies a minimum value of 1 for an ascending sequence and -(10)^26 for a descending sequence.

MAXVALUE: Specifies the maximum value that a sequence can generate.

NOMAXVALUE: Specifies a maximum of 10^27 for an ascending sequence or -1 for a descending sequence. This is the default clause.

START WITH: Specifies the first sequence number to be generated. The default for an ascending sequence is the sequence minimum value (1) and for a descending sequence, it is the maximum value (-1).

CYCLE: Specifies that the sequence continues to generate repeat values after reaching either its maximum value.

NOCYCLE: Specifies that a sequence cannot generate more values after reaching the maximum value.

CACHE: Specifies how many values of a sequence Oracle pre-allocates and keeps in memory for faster access. The minimum value for this parameter is **two.**

NOCACHE: Specifies that values of a sequence are not pre-allocated.

Note

If the CACHE / NOCACHE clause is omitted ORACLE caches 20 sequence numbers by default.

ORDER: This guarantees that sequence numbers are generated in the order of request. This is necessary if using Parallel Server in Parallel mode option. In exclusive mode option, a sequence always generates numbers in order.

NOORDER: This does not guarantee sequence numbers are generated in order of request. This is necessary if you are using Parallel Server in Parallel mode option. If the ORDER/NOORDER clause is omitted, a sequence takes the NOORDER clause by default.

Note

The ORDER, NOORDER Clause has no significance, if Oracle is configured with Single Server option.

Example 20:

Create a sequence by the name **ADDR_SEQ**, which will generate numbers from 1 upto 9999 in ascending order with an interval of 1. The sequence must restart from the number 1 after generating number 999.

```
CREATE SEQUENCE ADDR_SEQ INCREMENT BY 1 START WITH 1
   MINVALUE 1 MAXVALUE 999 CYCLE;
```

Referencing A Sequence

Once a sequence is created SQL can be used to view the values held in its cache. To simply view sequence value use a SELECT sentence as described below:

```
SELECT <SequenceName>.NextVal FROM DUAL;
```

This will display the next value held in the cache on the VDU screen. Every time **nextval** references a sequence its output is automatically incremented from the old value to the new value ready for use.

The example below explains how to access a sequence and use its generated value in the INSERT statement.

Example 21:

Insert values for ADDR_TYPE, ADDR1, ADDR2, CITY, STATE and PINCODE in the **ADDR_DTLS** table. The **ADDR_SEQ** sequence must be used to generate ADDR_NO and CODE_NO must be a value held in the BRANCH_NO column of the BRANCH_MSTR table.

Table Name: ADDR_DTLS

Column Name	Data Type	Size	Attributes
ADDR_NO	Number	6	Primary Key
CODE_NO	VarChar2	10	Foreign Key references BRANCH_NO of the BRANCH_MSTR table.
ADDR_TYPE	VarChar2	1	Can hold the values: **H** for Head Office or **B** for Branch

Column Name	Data Type	Size	Attributes
ADDR1	VarChar2	50	
ADDR2	VarChar2	50	
CITY	VarChar2	25	
STATE	VarChar2	25	
PINCODE	VarChar2	6	

```
INSERT INTO ADDR_DTLS (ADDR_NO, CODE_NO, ADDR_TYPE, ADDR1, ADDR2, CITY,
STATE, PINCODE) VALUES(ADDR_SEQ.NextVal, 'B5', 'B', 'Vertex Plaza, Shop 4,', 'Western Express
Highway, Dahisar (East),', 'Mumbai', 'Maharashtra', '400078');
```

To reference the current value of a sequence:

```
SELECT <SequenceName>.CurrVal FROM DUAL;
```

This is a method a numeric value generated by the system, using a sequence can be used to insert values into a primary key column.

The most commonly used technique in commercial application development is to concatenate a sequence generated value with a user-entered value.

The ADDR_NO stored in the **ADDR_DTLS** table, can be a concatenation of the month and year from the system date and the number generated by the sequence ADDR_SEQ. For example ADDR_NO 01041 is generated with 01 (month in number format), 04 (year in number format) and 1(a sequence generated value).

To help keep the sequence-generated number from becoming too large, each time either the month (or year) changes the sequence can be reset.

The sequence can be reset at the end of each month. If the company generated 50 addresses are keyed in for the month of January 2004, the ADDR_DTLS will start with 01041 upto 010450. Again when the month changes to February and as the sequence is reset, the numbering will start with 02041, 02042 and so on.

Using this simple technique of resetting the sequence at the end of each month and concatenating the sequence with the system date, unique values can be generated for the ADDR_NO column and reduce the size of the number generated by the sequence.

Example 22:

INSERT INTO ADDR_DTLS (ADDR_NO, CODE_NO, ADDR_TYPE, ADDR1, ADDR2, CITY, STATE, PINCODE) **VALUES(TO_CHAR(SYSDATE, 'MMYY) || TO_CHAR(ADDR_SEQ.**NextVal), 'B5', 'B', 'Vertex Plaza, Shop 4,', 'Western Express Highway, Dahisar (East),', 'Mumbai', 'Maharashtra', '400078');

Altering A Sequence

A sequence once created can be altered. This is achieved by using the **ALTER SEQUENCE** statement.

Syntax:

```
ALTER SEQUENCE <SequenceName>
    [INCREMENT BY <IntegerValue> MAXVALUE <IntegerValue> / NOMAXVALUE
    MINVALUE <IntegerValue> / NOMINVALUE CYCLE / NOCYCLE
    CACHE <IntegerValue> / NOCACHE ORDER / NOORDER]
```

Note

The **START** value of the sequence cannot be altered.

Example 23:

Change the Cache value of the sequence ADDR_SEQ to 30 and interval between two numbers as 2.

ALTER SEQUENCE ADDR_SEQ **INCREMENT BY** 2 **CACHE** 30;

Dropping A Sequence

The **DROP SEQUENCE** command is used to remove the sequence from the database.

Syntax:

```
DROP SEQUENCE <SequenceName>;
```

Example 24:
Destroy the sequence **ADDR_SEQ**.

DROP SEQUENCE ADDR_SEQ;

SNAPSHOTS

A snapshot is a recent **copy** of a table from database or in some cases, a subset of rows/columns of a table. The SQL statement that creates and subsequently maintains a snapshot normally reads data from a database residing on the server. A snapshot is created on the destination system with the **create snapshot** SQL command. The **remote table** is immediately defined and populated from the **master** table.

In a distributed computing environment, the snapshots are defined considering the following reasons:

- Response time improves when a local read-only copy of a table exists – this can be many times faster than reading data directly from a remote database.
- Once a snapshot is built on a remote database, if the node containing the data from which the snapshot is built is not available, the snapshot can be used without the need to access the unavailable database.

The query that creates the snapshot closely resembles the code used to create a view. The secret to keep a snapshot up to date is the specification of it's **refresh interval**. When defining a snapshot, the DBA specifies this interval, and Oracle 8i from then on automatically manages the propagation of data from the table(s) upon which the snapshot is built.

Snapshots are used to dynamically replicate data between distributed databases. The master table will be updateable but the snapshots can be either read-only or updateable. Read-only snapshots are the most common types of snapshots implemented. There are two types of snapshots available i.e. complex snapshots and simple snapshots.

In a simple snapshot, each row is based on a single row in a single remote table. A row in a complex snapshot may be based on more than one row in a remote table, such as via a **group by** operation or on the result of a **multi-table join**. Simple snapshots are thus a subset of the snapshots that can be created.

Creating A Snapshot

Snapshots can be simple or complex. A simple snapshot consists pf either single table or a simple SELECT of rows from a single table. A complex snapshot consists of joined tables, views, or grouped and complex SELECT statements queries.

Syntax:

```
CREATE SNAPSHOT <SnapshotName>
ALTER SNAPSHOT <SnapshotName>
   [<schema>]
   [  [PCTFREE <Integer>] [PCTUSED <Integer>] [INITRANS <Integer>]
      [MAXTRANS <Integer>] [TABLESPACE <Tablespace>]
      [STORAGE <StorageClause>]
   /  [CLUSTER <Cluster> (<Column1>[,<Column2>, ...])]
   ]
   [USING
```

```
        [INDEX
            [PCTFREE <Integer>] [PCTUSED <Integer>]
            [INITRANS <Integer>] [MAXTRANS <Integer>]
        ]
        [DEFAULT ROLLBACK SEGMENT
            [MASTER <RollbackSegment>/LOCAL]
        ]
    ]
    [REFRESH [FAST/COMPLETE/FORCE]
        [START WITH <Date>] [NEXT <Date>]
        [WITH [PRIMARY KEY/ROWID]]
    ]
    AS <SubQuery>
    [FOR UPDATE]
```

The keywords and parameters for the CREATE SNAPSHOT are as follows:

- **schema** – Contains the snapshot. If not specified, Oracle creates the snapshot in user's schema.
- **snapshot** – Specifies the name of the snapshot to be created. Oracle chooses names for the table, views and index used to maintain the snapshot by adding a prefix and suffix to the snapshot name. To limit these names to 30 bytes and allow them to contain the entire snapshot name, limit your snapshot names to 19 bytes.
- **PCTFREE, PCTUSED, INITRANS, and MAXTRANS** – Establish values for the specified parameters for the internal table Oracle uses to maintain the snapshot's data.
- **TABLESPACE** – Specifies the tablespace in which the snapshot is to be created. If user omits this option, Oracle creates the snapshot in the default tablespace of the owner of the snapshot's schema.
- **STORAGE** – Establishes storage characteristics for the table Oracle uses to maintain the snapshot's data.
- **CLUSTER** – Creates the snapshot as part of the specified cluster. Because a clustered snapshot uses the cluster's space allocation, **do not use** the **PCTFREE, PCTUSED, INITRANS, MAXTRANS, TABLESPACE**, or **STORAGE** parameters with the CLUSTER option.
- **USING INDEX** – Specifies parameters for the index Oracle creates to maintain the snapshot. User can choose the values of the **INITRANS, MAXTRANS, TABLESPACE, STORAGE**, and **PCTFREE** parameters. For the **PCTFREE, PCTUSED, INITRAANS**, and **MAXTRANS** parameters, specify the default storage and transaction attributes for the snapshot.
- **ROLLBACK SEGMENT** – Specifies the local snapshot and/or remote master rollback segments to be used during snapshot refresh.
- **DEFAULT** – Specifies that Oracle will choose which rollback segment to use.
- **MASTER** – Specifies the rollback segment to be used at the remote master for the individual snapshot.
- **LOCAL** – Specifies the rollback segment to be used for the local refresh group that contains the snapshot. If user does not specify **MASTER** or **LOCAL**, Oracle uses **LOCAL by default**. If user does not specify rollback segment, Oracle chooses the rollback segment to be used automatically. If user specifies **DEFAULT**, user **cannot specify rollback** segment.
- **REFRESH** – Specifies how and when Oracle automatically refreshes the snapshot.
- **FAST** – Specifies a fast refresh or one using only the updated data stored in the snapshot log associated with the master table.

- **COMPLETE** – Specifies a complete refresh or one that re-executes the snapshot's query.
- **FORCE** – Specifies a fast refresh, if one is possible, or complete refresh, if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time. If user omits the **FAST**, **COMPLETE**, and **FORCE** options, Oracle uses FORCE by default.
- **START WITH** – Specifies a date expression for the first automatic refresh time.
- **NEXT** – Specifies a date expression for calculating the interval between automatic refreshes. Both the **START WITH** and **NEXT** values must evaluate to a time in the future. If user omits the **START WITH** value, Oracle determines the first automatic refresh time by evaluating the **NEXT** expression when a user creates the snapshot. If a user specifies a START WITH value but omits the **NEXT** value, Oracle refreshes the snapshot only once. If a user omits both the **START WITH** and **NEXT** values, or if you omit the **REFRESH** clause entirely, Oracle does not automatically refresh the snapshot.
- **WITH PRIMARY KEY** – Specifies that primary key snapshots are to be created. These snapshots allow snapshot master tables to be reorganized without impacting the snapshot's ability to continue to fast refresh. A user can also define primary key snapshots as simple snapshots with subqueries.
- **WITH ROWID** – Specifies that **ROWID** snapshots are to be created. These snapshots provide backward compatibility with Oracle Release 7.0 masters. If a user omits both **WITH PRIMARY KEY** and **WITH ROWID**, Oracle creates primary key snapshots by default.
- **FOR UPDATE** – Allows a simple snapshot to be updated. When used in conjunction with the replication option, these updates will be propagated to the master.
- **AS <subquery>** – Specifies the snapshot query. When a user creates the snapshot, Oracle executes this query and places the results in the snapshot. The select list can contain up to 1,000 expressions. The syntax of a snapshot query is described with the syntax description of a subquery. The syntax of a snapshot query is subject to the same restrictions as a view query. For a list of these restrictions, see the **CREATE VIEW** command.

Example 25:
Following code creates a snapshot of **EMP_MSTR** table in the default tablespace i.e. System

```
CREATE SNAPSHOT NEW_EMP
    PCTFREE 10 PCTUSED 70 TABLESPACE System
    STORAGE (INITIAL 50K NEXT 50K PCTINCREASE 0)
    REFRESH START WITH ROUND(SYSDATE + 7) + 2/24
        NEXT NEXT_DATE(TRUNC(SYSDATE, 'MONDAY') + 2/24
    AS SELECT * FROM EMP_MSTR;
```

Altering A Snapshot

A snapshot is altered using ALTER SNAPSHOT command. Usually storage and space usage parameters, types and frequency of refresh are altered.

Syntax:

```
ALTER SNAPSHOT <SnapshotName>
    [<Schema>]
    [   [PCTFREE <Integer>] [PCTUSED <Integer>]
        [INITRANS <Integer>] [MAXTRANS <Integer>]
        [TABLESPACE <Tablespace>] [STORAGE <StorageClause>]
    /   [CLUSTER <Cluster> (<Column1>[,<Column2>, ...])]
    ]
```

```
[USING
    [INDEX
        [PCTFREE <Integer>] [PCTUSED <Integer>]
        [INITRANS <Integer>] [MAXTRANS <Integer>]
    ]
    [DEFAULT ROLLBACK SEGMENT
        [MASTER <rollback segment>/LOCAL]
    ]
]
[REFRESH [FAST/COMPLETE/FORCE]
    [START WITH <date>] [NEXT <date>] [WITH PRIMARY KEY]
]
```

The parameters for ALTER SNAPSHOT are as follows:

- ❑ **Schema** – is the schema in which to store the log. If not specified, this will default to the user's own schema.
- ❑ **<Table>** – is the table name to create the snapshot log for.
- ❑ **PCTFREE**, **PCTUSED**, **INITRANS**, and **MAXTRANS** – are the values for these creation parameters to use for the created log file.
- ❑ **STORAGE** – is a standard storage clause.
- ❑ **REFRESH** – specifies the refresh mode:
- ❑ **FAST** – uses a **SNAPSHOT LOG**. It is the default mode.
- ❑ **COMPLETE** – re-performs the sub-query and is the only valid mode for a complex snapshot.
- ❑ **FORCE** – causes the system to first try a **FAST**, and if this is not possible, then a **COMPLETE**.
- ❑ **START WITH** – specifies the date for the first refresh.
- ❑ **NEXT** – specifies either a date or a time interval for the next refresh of the snapshot. **START WITH** and **NEXT** values are used to determine the refresh cycle for the snapshot. If just **START WITH** is specified, only the initial refresh is done. If both are specified, the first is done on the **START WITH** date, and the **NEXT** is evaluated against the **START WITH** to determine future refreshes. If just the **NEXT** value is specified, if computers based on the date the snapshot is created. If neither is specified, the snapshot is not automatically refreshed.

Dropping A Snapshot

A snapshot is dropped using DROP SNAPSHOT command.

Syntax:

```
DROP SNAPSHOT <SnapshotName>
```

Example 26:

DROP SNAPSHOT New_Client

Note

When a snapshot is dropped, if it has a snapshot log associated with, only the rows required for maintaining that snapshot are dropped. Dropping a master table upon which a snapshot is based does not drop the snapshot. Any subsequent refreshes however, will fail.

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. ________ a table is an access strategy, that is, a way to sort and search records in the table.
2. Indexing involves forming a ________ dimensional matrix completely independent of the table on which the index is being created.
3. The address field of an index is called ________.
4. Each data file is further divided into ________ and each block is given a unique number.
5. An index that allows duplicate values for the indexed columns is called ________ indexes.
6. An index created on a single column of a table is called ________ Index.
7. An index created on more than one column it is called ________ Index.
8. If an index is created on more than one column it is called ________ Index.
9. A reverse key index can be rebuilt into a normal index using the keywords ________.
10. ________ indexes are typically only a fraction of the size of the indexed data in the table.
11. An index is ________ prefixed if it is partitioned on the left prefix of the index columns.
12. In case of a ________ Index, the index cannot be split or merged separately.
13. Querying ________ will provide details on columns on which the user's indexes are created.
14. A ________ is an SQL statement that extracts values from table columns using a SELECT statement and passes these values as input to another SQL statement.
15. The SQL statement to which the values of the select statement are passed is called ________ statement.
16. To reduce redundant data to the minimum possible, an object is created called a ________.
17. A View that is used to Look at table data as well as Insert, Update and Delete table data is called an ________ View.
18. The concept of a ________ is where member records are stored physically near parent records.
19. The columns within the cluster index are called the ________.
20. ________ that are updated often are not good candidates for the cluster key.
21. The ________ parameter specifies the interval between sequence numbers.
22. The ________ parameter specifies a minimum value of 1 for an ascending sequence and -(10)^26 for a descending sequence.
23. The ________ parameter specifies the maximum value that a sequence can generate.
24. The ________ parameter specifies that values of a sequence are not pre-allocated.

25. In Snapshots ____________________ improves when a local read-only copy of a table exists.

26. A ____________ snapshot consists of joined tables, views, or grouped and complex SELECT statement queries.

27. A ____________ snapshot consists of either single table or a simple SELECT of rows from a single table.

28. The ____________ parameter establishes storage characteristics for the table Oracle uses to maintain the snapshot's data.

29. The ____________ parameter creates the snapshot as part of the specified cluster.

30. The ____________________ parameter specifies the local snapshot and/or remote master rollback segments to be used during snapshot refresh.

31. The ____________ parameter specifies the rollback segment to be used at the remote master for the individual snapshot.

32. In a CREATE SNAPSHOT the ____________ parameter specifies a fast refresh, if one is possible, or complete refresh, is a fast refresh is not possible.

33. In a CREATE SNAPSHOT the ____________ parameter specifies that ROWID snapshots are to be created.

34. A user can also define primary key snapshots as simple snapshots with ____________

35. A snapshot is altered using ____________________ command.

36. In ALTER SNAPSHOT the ____________ parameter re-performs the subquery and is the only valid mode for a complex snapshot.

TRUE OR FALSE

37. Indexes adversely affect the speed at which the records are retrieved.

38. An index is an ordered list of the contents of a row, (or a group of rows) of a table.

39. ROWID is an internally generated and maintained, binary value, which identifies a record.

40. Each data block can store only one Record.

41. Data files are the files used by the Oracle engine to store user data.

42. Indexes that deny duplicate values for the indexed columns are called Unique Indexes.

43. An index cannot be created on more than one column.

44. Each data file is given an unique number at the time of data file creation.

45. If an index is created on a single column it is called Single Unique Index.

46. To create unique index, the keyword UNIQUE should be included in the Create Index command.

47. A normal index can be rebuilt as a reverse key index.

48. In adhoc queries or similar situations, bitmap indexes adversely affect the performance of queries.

49. A column's index will not be used when the same column is expressed in an arithmetic expression or function in the WHERE clause.

50. The expression cannot contain any aggregate functions.

51. When key compression is used, the duplicate key will be stored as a suffix entry on the index block.

52. It is not possible to manipulate index partitions.

53. Querying USER_INDEXES will provide details on INDEXES that the user has created.

54. Indexes associated with the tables can be removed by using the DELETE INDEX command.

55. Indexes speeds up data retrieval.

56. The SELECT statement is called Parent SQL statement.

57. The order in which data is retrieved is dependent upon the indexes created on the table.

58. The ORDER BY clause cannot be used while creating a view.

59. The DELETE or MODIFY operations affect the Master table.

60. Clusters are used to store data from different tables in the same physical data blocks.

61. Clusters can reduce the performance of UPDATE statements as compared with storing a table separately with its own index.

62. The INCREMENT BY parameter can be any positive or negative value or zero.

63. The MINVALUE specifies the sequence minimum value.

64. The CYCLE parameter specifies that the sequence continues to generate repeat values after reaching its minimum value.

65. The NOMAXVALUE parameter specifies a maximum of 10^27 for an ascending sequence or -1 for a descending sequence.

66. The NOCYCLE keyword specifies that a sequence can generate more values after reaching the maximum value.

67. Snapshots are used to dynamically replicate data between distributed databases.

68. PCTFREE parameter for the CREATE SNAPSHOT establishes values for the specified parameters for the internal table Oracle uses to maintain the snapshot's data.

69. The DEFAULT parameter specifies the rollback segment to be used at the remote master for the individual snapshot.

70. If a user omits both WITH PRIMARY KEY and WITH ROWID, Oracle creates primary key snapshots by default.

71. In a CREATE SNAPSHOT the DATE parameter specifies a date expression for the first automatic refresh time.

72. In ALTER SNAPSHOT the FORCE parameter causes the system to first try a FAST, and if this is not possible, then a COMPLETE.

HANDS ON EXERCISES

1. Write appropriate SQL statements for the following:

a) Create a simple index idx_Prod on product cost price from the Product_Master table.

b) Create a sequence inv_seq with the following parameters, increment by 3, cycle, cache 4 and which will generate the numbers from 1 to 9999 in ascending order.

c) Create view on OrderNo, OrderDate, OrderStatus of the Sales_Order table and ProductNo,ProductRate and QtyOrdered of Sales_Order_Details.