Database Management

Excellent Book keeping strategies existed before the computer age, on the management of data, including convenient and efficient retrieval and update operations. But, because of the limitations associated with the physical handling of documents and human processing, a look in retrospect suggests that data (and information generated through data) were only marginally used in daily decision-making, even in progressive organizations.

The application of computers speeds up the data processing activities and adds some flexibility to the management of data and the information generated from it. Widespread and easy access to time sharing systems, advances in logical and physical access methods matched by steep increases in the density and volume of disk storage devices, led gradually to a reassessment of the role of data in management decision making.

This transition was facilitated by the induction of database technology into the organizations.

Approaches to Data Management

Manual methods of Data Management

- Convenient and efficient retrieval
- Updating Operations

Limitation of Manual Data Management

- Physical Volume Data
- Human Processing of Data

Technological Advancement in Data Management

- Using Computers to speedup processing of Data
- Advancement of Processing Power
- Using time-sharing for multiple users
- High-speed for multiple users
- Centralized to distributed databases
- Centralized to distributed Processing (Client/Server)

Database Management System

The evolution of database management was accompanied and promoted by Advances in computing: Hardware, OS and Networking
Drawbacks of the prevalent approach to data management

- Data Redundancy
- Risk to data integrity
- Data isolation
- Difficult access to data
- Unsatisfactory security measures
- Poor support of concurrent access of data

Before knowing about **Oracle** we must know about what is **Database Management System** (**DBMS**). A Database Management System is essentially a collection of interrelated data and a set of programs to access this data. This collection of data is called Database Management System. The primary objective of a DBMS is to provide a convenient environment to retrieve and store database information. Database Management System supports only Single user environment.

Database Models

- Hierarchical
- Network
- Relational

Hierarchical Model

This model was introduced in the Information Management System (IMS) developed by IBM in 1968. This model is like a hierarchical tree structure, used to construct a hierarchy of records in the form of nodes and branches. The data elements present in the structure have Parent-Child relationship. Closely related information in the parent-child structure is stored together as a logical unit. A parent unit may have many child units, but a child is restricted to have only one parent.

The drawbacks of this model are :

- The hierarchical structure is not flexible to represent all the relationship proportions, which occur in the real world.
- It cannot demonstrate the overall data model for the enterprise because of the non-availability of actual data at the time of designing the data model.
- It cannot represent the Many-to-Many relationship.

Network Model

It supports the One-To-One and One-To-Many types only. The basic objects in this model are Data Items, Data Aggregates, Records and Sets.

It is an improvement on the Hierarchical Model. Here multiple parent-child relationships are used. Rapid and easy access to data is possible in this model due to multiple access paths to the data elements.

Relational Model

- Does not maintain physical connection between relations
- Data is organized in terms of rows and columns in a table
- The position of a row and/or column in a table is of no importance
- The intersection of a row and column must give a single value
- Features of an RDBMS
    - o The ability to create multiple relations and enter data into them
    - o An attractive query language
    - o Retrieval of information stored in more than one table
- An RDBMS product has to satisfy at least Seven of the 12 rules of Codd to be accepted as a full-fledged RDBMS.

Relational Database Management System

RDBMS is acronym for Relation Database Management System. **Dr. E. F. Codd** first introduced the Relational Database Model in 1970. The Relational model allows data to be represented in a simple row-column. Each data field is considered as a column and each record is considered as a row. Relational Database is more or less similar to Database Management System. In relational model there is relation between their data elements. Data is stored in tables. Tables have columns, rows and names. Tables can be related to each other if each has a column with a common type of information. The most famous RDBMS packages are Oracle, Sybase and Informix.
Simple example of Relational model is as follows :

Student Details Table

Roll_no	Sname	S_Address
1	Rahul	Satelite
2	Sachin	Ambawadi
3	Saurav	Naranpura

Student Marksheet Table

Rollno	Sub1	Sub2	Sub3
1	78	89	94
2	54	65	77
3	23	78	46

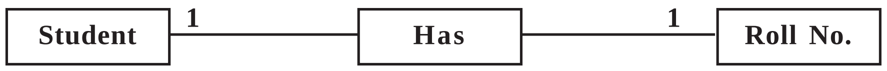
Here, both tables are based on students details. Common field in both tables is *Rollno*. So we can say both tables are related with each other through *Rollno* column.

Degree of Relationship

- One to One (1:1)
- One to Many or Many to One (1:M / M: 1)
- Many to Many (M: M)

The **Degree of Relationship** indicates the link between two entities for a specified occurrence of each.

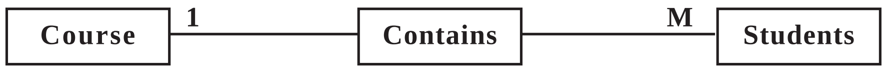
One to One Relationship : (1:1)



One student has only one Rollno

For one occurrence of the first entity, there can be, at the most one related occurrence of the second entity, and vice-versa.

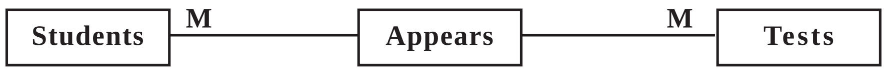
One to Many or Many to One Relationship: (1:M/M: 1)



As per the Institutions Norm, One student can enroll in one course at a time however, in one course, there can be more than one student.

For one occurrence of the first entity there can exist many related occurrences of the second entity and for every occurrence of the second entity there exists only one associated occurrence of the first.

Many to Many Relationship: (M:M)



The major disadvantage of the relational model is that a clear-cut interface cannot be determined. Reusability of a structure is not possible. The Relational Database now accepted model on which major database system are built. Oracle has introduced added functionality to this by incorporated object-oriented capabilities. Now it is known is as Object Relational Database Management System (ORDBMS). Object-oriented concept is added in Oracle8.

Some basic rules have to be followed for a DBMS to be relational. They are known as Codd's rules, designed in such a way that when the database is ready for use it encapsulates the relational theory to its full potential. These twelve rules are as follows.

E. F. Codd Rules

- The Information Rule
  All information must be store in table as data values.
- The Rule of Guaranteed Access
  Every item in a table must be logically addressable with the help of a table name.
- The Systematic Treatment of Null Values
  The RDBMS must be taken care of null values to represent missing or inapplicable information.
- The Database Description Rule
  A description of database is maintained using the same logical structures with which data was defined

by the RDBMS.

- Comprehensive Data Sub Language
  According to the rule the system must support data definition, view definition, data manipulation, integrity constraints, authorization and transaction management operations.
- The View Updating Rule
  All views that are theoretically updateable are also updateable by the system.
- The Insert and Update Rule
  This rule indicates that all the data manipulation commands must be operational on sets of rows having a relation rather than on a single row.
- The Physical Independence Rule
  Application programs must remain unimpaired when any changes are made in storage representation or access methods.
- The Logical Data Independence Rule
  The changes that are made should not affect the user's ability to work with the data. The change can be splitting table into many more tables.
- The Integrity Independence Rule
  The integrity constraints should store in the system catalog or in the database.
- The Distribution Rule
  The system must be access or manipulate the data that is distributed in other systems.
- The Non-subversion Rule
  If a RDBMS supports a lower level language then it should not bypass any integrity constraints defined in the higher level.

Object Relational Database Management System

Oracle8 and later versions are supported object-oriented concepts. A structure once created can be reused is the fundamental of the OOP's concept. So we can say Oracle8 is supported Object Relational model, Object – oriented model both.

Oracle products are based on a concept known as a client-server technology. This concept involves segregating the processing of an application between two systems. One performs all activities related to the database (server) and the other performs activities that help the user to interact with the application (client). A client or front-end database application also interacts with the database by requesting and receiving information from database server. It acts as an interface between the user and the database. The database server or back end is used to manage the database tables and also respond to client requests.

Client/Server Overview

- Client/Server architecture splits an application into tasks that are executed as separate processes
- Client/Server architecture is designed to assign tasks to the most appropriate processor
- Diverse areas of IT are involved, such as
    - o User Interface
    - o Database Programming

- Networking
- Distributed Data and Distributed Processing

Definition of Client/Server Computing

In Client/Server computing the application and data are distributed over Client System(s) sending requests and Server System(s) serving the request over a network.

The Client (often referred as the front-end) in a Client/Server model is a(ny) hardware platform, which handles the functionality of user interface, user request and user presentation. It may also have some local data and local processing.

The Server in a Client/Server model (the back-end) is a(ny) hardware platform, which handles requests and interface to a database.

Key Benefits of Client/Server Architecture

- Empowering Users :
  - User driven applications
  - Better User interface and utilization of resources
  - Intelligent workstations provide good interaction with the users
- Interoperability:
  - High-power, inexpensive servers provide data repository and processing functions.
  - Can be designed to offer open architecture (open connectivity and protocol)
- Extensibility:
  - Modular processor can be sized to match requirements
  - Advantage of latest technology
- Cost Advantages:
  - Increased productivity due to better technological response
  - Less operational cost of Client/Server system
  - High processing power of Client work-stations
  - Wider range of open and economic products
  - Better location of data
  - Better mapping of data

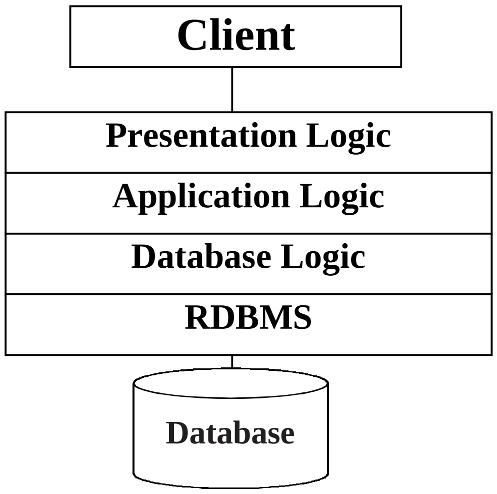
Client/Server solutions are based on distributed data and processing. As a result, the data resides where it logically belongs. Each Client owns its local data and has a secured access to a central database.

As desktops are becoming powerful and plenty, it makes economic sense to distribute the Client functionality on cheaper desktops. As a direct consequence, it frees some of the costly Server capability.

Latest techniques support Client/Server computing such as Rapid Application Development, Information Modeling and GUI. These tools and techniques increase programmer and user productivity. The ability to add servers to meet performance and geographical demand, without having to redesign the whole system, is another major advantage of a Client/Server system.

Distributed Processing in Client/Server

- Presentation Logic : User Interface, Client Application Logic
- Application Logic : Business rules
- Database Logic
- RDBMS



When major application components are distributed, the link between them must be transparent, with well-defined boundaries and protocols.

The Role of an Application Program Interface (API) is to ensure a formal, standard and clean interface between the application components.

- Manage Windows
- Manage dialogues
- Do local processing
- Manage presentations
- Interface with utilities
- Apply external controls
- Interface with Server(s)

The Client and Server's Role in Distributed Processing

The Server's Role in Distributed Processing

- ➢ Directory Management
- ➢ Query Management
- ➢ Database Administration Support
- ➢ Integrity and Security
- ➢ Concurrency Control
- ➢ Recovery Management
- ➢ Backup the Database(s)
- ➢ Performance Monitoring

The Client's Role in Distributed Processing

- ➢ Manage Windows
- ➢ Manage Dialogues
- ➢ Do Local Processing
- ➢ Manage Presentations
- ➢ Interface with Utilities
- ➢ Apply External Controls
- ➢ Interface with Server(s)

Introduction to ORACLE

ORACLE is a powerful RDBMS product that provides efficient and effective solutions for major database features. This includes:

- Large databases and space management control
- Many concurrent database users
- High transaction processing performance
- High availability
- Controlled availability
- Industry accepted standards
- Manageable security
- Database enforced integrity
- Client/Server environment
- Distributed database systems
- Portability
- Compatibility
- Connectivity

An ORACLE database system can easily take advantage of distributed processing by using its Client/ Server architecture. In this architecture, the database system is divided into two parts:

A front-end or a client portion
The client executes the database application that accesses database information and interacts with the user.

A back-end or a server portion
The server executes the ORACLE software and handles the functions required for concurrent, shared data access to ORACLE database.

What is SQL and SQL*Plus

Oracle was the first company to release a product that used the English-based Structured Query Language or SQL. This language allows end users to manipulate information of table(primary database object). To use SQL you need not to require any programming experience. SQL is a standard language common to all relational databases. SQL is database language used for storing and retrieving data from the database. Most Relational Database Management Systems provide extension to SQL to make it easier for application developer.

A table is a primary object of database used to store data. It stores data in form of rows and columns.

SQL*Plus is an Oracle tool (specific program) which accepts SQL commands and PL/SQL blocks and executes them. SQL *Plus enables manipulations of SQL commands and PL/SQL blocks. It also performs additional tasks such as calculations, store and print query results in the form of reports, list column definitions of any table, access and copy data between SQL databases and send messages to and accept responses from the user. SQL *Plus is a character based interactive tool, that runs in a GUI environment. It is loaded on the client machine.

To communicate with Oracle, SQL supports the following categories of commands:

1. **Data Definition Language**
   Create, Alter, Drop and Truncate
2. **Data Manipulation Language**
   Insert, Update, Delete and Select
3. **Transaction Control Language**
   Commit, Rollback and Savepoint
4. **Data Control Language**
   Grant and Revoke

Before we take a look on above-mentioned commands we will see the data types available in Oracle.

Oracle Internal Data types

When you create a table in Oracle, a few items should be important, not only do you have to give each table a name(e.g. employee, customer), you must also list all the columns or fields (e.g. First_name, Mname, Last_name) associated with the table. You also have to specify what type of information that table will hold to the database. For example, the column Empno holds numeric information. An Oracle database can hold many different types of data.

Data type	Description
Char(Size)	Stores fixed-length character data to store alphanumeric values, with a maximum size of 2000 bytes. Default and minimum size is 1 byte.
Varchar2(Size)	Stores variable-length character data to store alphanumeric values, with maximum size of 4000 bytes.
Nchar(Size)	Stores fixed-length character data of length size characters or bytes, depending on the choice of national character set. Maximum size if determined by the number of bytes required storing each character with an upper limit of 2000 bytes. Default and minimum size is 1 character or 1 byte, depending on the character set.
Nvarchar2(Size)	Stores variable-length character string having maximum length size characters or bytes, depending on the choice of national character set. Maximum size is determined by the number of bytes required to store each character, with an upper limit of 4000 bytes.
Long	Stores variable-length character data up to 2GB(Gigabytes). Its lenth would be restricted based on memory space available in the computer.
Number [p,s]	Number having precision ***p*** and scale ***s***. The precision ***p*** indicates total number of digit varies from 1 to 38. The scale ***s*** indicates number of digit in fraction part varies from –84 to 127.
Date	Stores dates from January 1, 4712 B.C. to December 31, 4712 A.D. Oracle predefine format of Date data type is DD-MON-YYYY.
Raw(Size)	Stores binary data of length size. Maximum size is 2000 bytes. One must have to specify size with RAW type data, because by default it does not specify any size.
Long Raw	Store binary data of variable length up to 2GB(Gigabytes).

LOBS – LARGE OBJECTS

LOB is use to store unstructured information such as sound and video clips, pictures upto 4 GB size.

CLOB	A Character Large Object containing fixed-width multi-byte characters. Varying-width character sets are not supported. Maximum size is 4GB.
NCLOB	A National Character Large Object containing fixed-width multi-byte characters. Varying-width character sets are not supported. Maximum size is 4GB. Stores national character set data.
BLOB	To store a Binary Large Object such a graphics, video clips and sound files. Maximum size is 4GB.
BFILE	Contains a locator to a large Binary File stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4GB.

Apart from oracle internal data types, user can create their own data type, which is used in database and other database object. We will discuss it in the later part.

Data Definition Language

The data definition language is used to create an object, alter the structure of an object and also drop already created object. The Data Definition Languages used for table definition can be classified into following:

- Create table command
- Alter table command
- Truncate table command
- Drop table command

Creation of Table

Table is a primary object of database, used to store data in form of rows and columns. It is created using following command:

```
Create Table <table_name>
(column1 datatype(size), column2 datatype(size),
 ………,column(n) datatype(size)
);
```

Where, *table_name* is a name of the table and *coulumn1*, *column2* … *column n* is a name of the column available in *table_name* table.

☑ ***Each column is separated by comma.***

Pointes to be remember while creating a table.

- Table Name must be start with an alphabet.

- Table name and column name should be of maximum 30 character long.
- Column name should not be repeated in same table.
- Reserve words of Oracle cannot be used as a table and column name.
- Two different tables should not have the same name.
- Underscores, numerals and letters are allowed but not blank space or single quotes.

Example

```
   SQL> Create Table Student
   (rollno number(5),name varchar2(25),course_name varchar2(20),
    birth_date date, fees number(9,2)
   );
```

`Table Created.`

`SQL>Create Table Emp_Master(Empno number(5),ename varchar2(25),job varchar2(20),hiredate date,salary number(7),deptno number(4));`

`Table Created.`

☑ ***Semicolon is used as a terminator. Every SQL command ends with a semicolon. Without semicolon compiler won't execute the command.***

Above definition will create simple table. Still there are more additional option related with create table for the object-relation feature we will discuss it afterwards.

Desc command

Describe command is external command of Oracle. The describe command is used to view the structure of a table as follows.

```
     Desc <table name>
```

Example

```
SQL> desc emp_master

 Name        Null?      Type
 -------     ----       ------
 EMPNO                  NUMBER(5)
 ENAME                  VARCHAR2(25)
 JOB                    VARCHAR2(20)
 HIREDATE               DATE
 SALARY                 NUMBER(7)
 DEPTNO                 NUMBER(4)
```

Alteration of Table

Once Simple Table is created, if there is a need to change the structure of a table at that time alter command is used. It is used when a user want to add a new column or change the width of datatype or datatype itself or to add or drop integrity constraints or column. (we will see about constraints very soon.).

i.e. table can be altered in one of three way : by adding column, by changing column definition or by dropping column.

Addition of Column(s)

Addition of column in table is done using:

```
Alter table <table_name> add(column1 datatype, column2 datatype ………);
```

Add option is used with "alter table" when you want to add a new column in existing table. If you want to Add more than one column then just write column name, data type and size in brackets. As usual Comma sign separates each column.

For Example, suppose you want to add column *comm* in *emp_master*, then you have to perform the following command.

```
   SQL>Alter Table Emp_master add comm number(7,2);
   Table altered.
```

If command performs successfully it will add comm. column in emp_master.

☑ ***We can add multiple columns in a single command.***

After adding column table structure is changed as per follows

```
SQL> desc emp_master

 Name       Null?     Type
 _______    ____      ______
 EMPNO                NUMBER(5)
 ENAME                VARCHAR2(25)
 JOB                  VARCHAR2(20)
 HIREDATE             DATE
 SALARY               NUMBER(7)
 DEPTNO               NUMBER(4)
 COMM                 NUMBER(7,2)
```

Deletion of Column

Till Oracle8 it is not possible to remove columns from a table but in *Oracle8i*, **drop** option is used with "alter table" when you want to drop any existing column.

```
Alter table <table_name> drop column <column name>;
```

Using above command you cannot drop more than one column at a time.

For Example, suppose you want to delete just before created column *comm* from the *emp_master*, then you have to apply following command.

```
SQL>Alter Table Emp_master drop column comm;
Table altered.
```

Dropping column is more complicated than adding or modifying a column, because of the additional work that Oracle has to do. Just removing the column from the list of columns in the table actually recover the space that was actually taken up by the column values that is more complex, and potentially very time-consuming for the database. For this reason, you can drop a column using unused clause. Column can be immediately remove column by drop clause, the action may impact on performance or one make marked column as unused using **unused caluse**, there will be no impact on performance. When unused caluse is used the column can actually be dropped at a later time when the database is less heavily used. One can marked column as a unused using :

`Alter table <table_name> set unused column <column name>;`

For Example,

```
SQL>Alter Table Emp_master set unused column comm;
Table altered.
```

Making a column as "unused" does not release the spcace previously used by the colum, until you drop the unused columns. It can be possible using:

`Alter table <table_name> drop unused columns;`

☑ ***Once you have marked column as "unused" you cannot access that column***

You can drop multiple columns at a time using single command as per follows

`Alter table <table_name> drop (Column1, Column2,…);`

The multiple columns name must be enclosed in parentheses.

Modification in Column

Modify option is used with "alter table" when you want to modify any existing column. If you want to modify data type or size of more than one column then just write column name, data type and size in brackets and each column is separated by comma sign as per follows:

`Alter table <table name> modify (column1 datatype,………);`

For Example, if you want to change size of *salary* column of *emp_master* the following command is performed.

```
SQL> Alter table emp_master  modify salary number(9,2);
Table altered.
```

It will change size of salary column from 7 to (9,2).

When you want to decrease the size of column, table must be empty. If table has any rows then it will not allow decrement in the column width.

Truncate Table

If there is no further use of records stored in a table and the structure is required then only data can be deleted using **Truncate** command. Truncate command will delete all the records permanently of specified table as follows.

```
Truncate table <table name> [Reuse Storage];
```

Example
Following command will delete all the records permanently from the table.

```
SQL>Truncate Table Emp_master;
```
Or

```
SQL>Truncate Table Emp_master Reuse Storage;
Table truncated.
```

Drop Table

If the table is no longer in use, drop command will drop table data and table structure both as follows:

```
Drop table <table name>;
```

For example, To drop emp_master table, the following command is used.

```
SQL>drop table emp_master;
Table dropped.
```

All the data definition commands are auto-committed. Once command is performed is cannot be rollback.

Constraints

Maintaining security and integrity of a database is the most important factor. Such limitations have to be enforced on the data, and only that data which satisfies the conditions will actually be stored for analysis. If the data gathered fails to satisfy the conditions set, it is rejected. This technique ensures that the data that is stored in the database will be valid, and has integrity. Rules, which are enforced on data being entered and prevents user from entering invalid data into tables are called **constraints**. Thus, constraints are super control data being entered in tables for permanent storage.

Oracle permits data constraints to be attached to table columns via SQL syntax that will check data for integrity. Once data constraints are part of a table column construction, the Oracle engine checks the data being entered into a table column against the data constraints. If the data passes this check, it is stored in the table, else the data is rejected. Even if a single column of the record being entered into the table fails a constraint, the entire record is rejected and not stored in the table. Both the “Create table” and “Alter table” SQL verbs can be used to write SQL sentences that attach constraints to a table column.

Until now we have created table without using any constraint, Hence the tables have not been given any instructions to filter what is being stored in the table.

The following are the types of integrity constraints

- Domain Integrity constraints
- Entity Integrity constraints
- Referential Integrity constraint

Oracle allows programmers to define constraints

- Column Level
- Table Level

Column Level constraints

If data constraints are defined along with the column definition when creating or altering a table structure, they are column level constraints. Column level constraints are applied to the current column. The current column is the column that immediately precedes the constraints i.e. they are local to a specific column. Column level constraints cannot be applied if the data constraints span across the multiple columns in a table.

Table Level Constraint

If the data constraints are defined after defining all the table columns when creating or altering a table structure, it is a table level constraint. Table Level constraints mostly used when data constraints spans across multiple columns in a table.

Domain Integrity Constraints

These constraints set a range and any violations that take place will prevent the user from performing the manipulations that caused the breached.

Domain integrity constraints are classified into two types

- Not Null constraint
- Check constraint

Not Null Constraint

Often there may be records in a table that do not have values for every field, either because the information is not available at the time of data entry or because field is not applicable in every case. Oracle will place a null value in the column in the absence of a user-defined value. By default every column will accept null values.

A **Null** values is different from a blank or a zero. We can say that Null means undefined. Null are treated specially by Oracle.

When a column is defined as not null, then that column becomes mandatory column. It implies that a value must be entered into the column. Remember that not null constraints can be applied on column level only.

Example

```
SQL> create table item_master
    (item_code varchar(5)  not null,
     item_name varchar(25) constraint n1 not null,
     item_rate  number(7,2),
     item_desc varchar(50));

Table created.
```

This declaration is specifying that first two columns will not accept null values.
In above command *n1* is a user defined constraint name of *item_name* column. While constraint name assigned internally to *item_code* column, because we omit user-defined constraint name to it.

One can add not null constraint in existing table as follows:

```
   SQL> alter table item_master modify item_desc not null;
```

While one can try to insert null value using

```
   insert into item_master(item_desc) values (null);
```

will cause following error

```
ERROR at line 1:
ORA-02296: cannot enable (SCOTT.ITEM_MASTER.ITEM_DESC) - null values
found
```

To drop not null constraint

Example:

```
SQL>Alter table item_master modify item_desc null;
```

Now item_desc column can accept Null values.

```
SQL> insert into item_master (item_desc) values (null);
```

1 row created.

Check Constraint

Business rule validations can be applied on a column using Check constraint. Check constraint must be specified by logical or boolean expressions.

Create a *client_master* table with following check constraints.

1. Data values being inserted into the column client_no must starts with the capital letter 'C'
2. Data values being inserted into the column name should be in upper case only.
3. Only allow "Mumbai" or "Ahmedabad" values for the city column.

Check constraint defined at column level as per follows:

```
SQL> create table client_master (client_no varchar(6) check client_no
like 'C%'), name varchar(20) check (name = upper(name)),address1
varchar(30), address2 varchar(30), city varchar(20) check (city in
('Mumbai','Ahemedabad'));
```

```
Table created.
```

Check constraint defined at table level as per follows:

```
SQL> create table client_master (client_no varchar(6), name varchar(20),
address1 varchar(30), address2 varchar(30),city varchar(20),check
(client_no like 'C%'),
check (name = upper(name)));

Table created.
```

Check constraint with alter command

```
SQL> alter table client_master add constraint chk_city check (city
('Mumbai','Ahmedabad'));

Table altered.
```

Entity Integrity Constraints

This type of constraints are further classified into

- Unique Constraint
- Primary Key Constraint

Unique Constraint

The purpose of unique key is to ensure that information in the column(s) is unique i.e. the value entered in column(s) defined in the unique constraint must not be repeated across the column. A table may have many unique keys. If unique constraint is defined in more than one column (combination of columns), it is said to be composite unique key. Maximum combination of columns that a composite unique key can contain is 16.

Example:

```
SQL> create table client_master (client_no varchar(6) Unique, name
varchar(20), address1 varchar(30), address2 varchar(30),city
varchar(20));
```

```
Table created.
```

Unique constraint defined at table level

Example:

```
SQL> create table client_master (client_no varchar(6), name varchar(20),
address1 varchar(30), address2 varchar(30),city varchar(20),
Unique(client_no));
                                OR
SQL> create table client_master (client_no varchar(6), name varchar(20),
address1 varchar(30), address2 varchar(30),city varchar(20),constraint
uni1 Unique (client_no));

Table created.
```

Unique constraint with alter command

Example:

```
SQL> Alter table client_master add constraint uni1 Unique (client_no);
```

Primary Key Constraint

A primary key is one or on more columns(s) in a table to uniquely identify each row in the table. A primary key column in a table has a special attribute. It defines the column, as a mandatory column i.e. the column cannot be left blank and should have a unique value. Here by default not null constraint is attached with the column.
A multicolumn primary key is called a Composite primary key. The only function of a primary key in a table is to uniquely identify a row. A table can have only one primary key.

Primary key constraint at the column level

Example:

```
SQL> create table order_master (order_no varchar(5) constraint pr_ono
primary key, order_dt date, vencode varchar(5), order_status char(1),
del_dt date);

Table created.

SQL> insert into order_master values (‘O001’,’20-jun-01’, ‘V001’,
‘c’,’22-jun-01’);

1 row created.

SQL> insert into order_master values (‘O001’,’20-jun-01’, ‘V001’,‘c’,’22-
jun-01’);
```

```
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.PR_ONO) violated
```

If we insert already inserted *order_no* the above mentioned error will generate.

```
SQL> insert into order_master values (null,’20-jun-01’, ‘V001’,‘c’,’22-
jun-01’);

ERROR at line 1:
ORA-01400 :cannot insert NULL into (“SCOTT”. “ORDER_MASTER” .”ORDER_NO”
)
```

If we try to insert null values it will generate error message.

Primary key defined at the table level

Example:

```
SQL> create table vendor_master (ven_code varchar(5), ven_name
varchar(20), venadd1 varchar(15), venadd2 varchar(15),vencity
varchar(15), constraint pr_ven primary key (ven_code));

Table created.
```

Composite Primary key defined at the table level: Composite Primary key is a primary key created with the combination of more than one key and combination values of both the fields should be unique

Example:

```
SQL> create table vendor_master (ven_code varchar(5), ven_name
varchar(20), venadd1 varchar(15), venadd2 varchar(15),vencity
varchar(15), constraint pr_com primary key (ven_code,ven_name));

Table created.
```

Primary key with alter command:

```
SQL> alter table emp_master add constraint pr_eno primary key (empno);

Table altered.
```

Referential Integrity Constraint

In this category there is only one constraint and it is Foreign Key & References

To establish a “parent-child” or a “master-detail” relationship between two tables having a common column, we make use of referential integrity constraint.

Foreign key represent relationships between tables. A foreign key is a column whose values are derived from the primary key or unique key. The table in which the foreign key is defined is called a foreign table or Detail table. The table that defines the primary or unique keys and is referenced by the foreign key is called the Primary table or Master table.

The master table can be referenced in the foreign key definition by using references keyword. If the column name is not specified, by default, Oracle references the primary key in the master table.

The existence of a foreign key implies that the table with the foreign key is related to the master table from which the foreign key is derived. A foreign key must have a corresponding primary key or a unique key value in a master table.

Principles of Foreign Key Constraint

- Rejects an insert or update of a value in a particular column, if a corresponding value does not exist in the master table.
- Deletion of rows from the Master table is not possible if detail table having corresponding values.
- Primary key or unique key must in Master table.
- Requires that the Foreign key column(s) and reference column(s) have same data type

References constraint defined at column level

Example:

```
SQL> create table order_detail (order_no varchar(5) constraint fk_ono
References order_master(order_no), item_code varchar(5), qty_ord
number(5), qty_deld number(5));
```

Table created.

The above command creates *order_detail* table with foreign key on *order_no* column, which refer *order_master* table's *order_no* column.

The referential integrity constraint does not use foreign key keyword to identify the columns that make up the foreign key. Because the constraint is defined at column level. The foreign key automatically enforced on the column.

Foreign key constraint defined at table level

Example:

```
SQL>create table order_detail (order_no varchar(5), item_code varchar(5),
qty_ord number(5), qty_deld number(5), constraint fk_ono references
order_master (order_no));
```

Table created.

We will insert records in order_detail table and see what happens.

```
SQL>insert into order_detail values (‘O001’, ’I001’ ,10 , 10);
1 row created.

SQL>insert into order_detail values (‘O002’, ’I001’ , 10, 10);

ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.FK_ONO) violated - parent key
not found
```

If we try to insert value, which is not available in parent table the above mention error occur.

Foreign Key Constraint with alter command

```
SQL> alter table order_detail add constraint fk_icode foreign key
(order_no) references order_master(order_no);

Table altered.
```

Remember that when we add constraint at table level foreign key keyword is must.

Suppose we try to delete record from order_master and corresponding values are available in order_detail (detail table) it will give an error.

```
SQL> delete from order_master where order_no = ‘O001’;

ERROR at line 1:
ORA-02292: integrity constraint (SCOTT.FK_ONO) violated - child record
found
```

On Delete Cascade clause

The on delete cascade option is used with foreign key. When the on delete cascade option is specified in the foreign key definition, if the user deletes a record in the master table, all the corresponding records in the detail table along with the master table record will be deleted.

Example:

```
SQL> alter table order_detail add constraint fk_icode foreign key
(item_code) references item_master(item_code) on delete cascade;

Table altered.
```

To drop constraint drop option is used with alter table command.

Syntax:

```
SQL> Alter Table <Table Name> Drop Constraint <Constraint Name>;
```

Example:

```
SQL> alter table order_detail drop constraint fk_icode;
Table altered.
```

Data Manipulation Command

Data Manipulation commands are most widely used SQL commands and they are

- Insert
- Update
- Delete
- Select

Insert command

After creation of table, it is necessary it should have data in it. The insert command is used to add data in form of one or more rows to a table as per follows:

`Insert into <table name> values(a list of data values);`

In a list of data values you have to specify values for each and every column in the same order as they are defined. A value of each column is separated by comma in the list.

The value of char, nchar, varchar2, nvarchar2, raw, long and date data types are enclosed in single quotes.

```
SQL> desc emp_master

 Name        Null?     Type
 -------     ----      ------
 EMPNO                 NUMBER(5)
 ENAME                 VARCHAR2(25)
 JOB                   VARCHAR2(20)
 HIREDATE              DATE
 SALARY                NUMBER(7)
 DEPTNO                NUMBER(4)
 COMM                  NUMBER(7,2)
```

Example

```
SQL>insert into emp_master values(1122, ’Allen’,  ‘Manager’, ‘1-jan-
00’,7500,10,1000);
1 row created.
```

Using insert command one can insert values in specific columns as follows:

`Insert into <table name>(column list) values(a list of data);`

Here number of column and a list of data should be same and list of data should be in order to column list.

```
SQL> insert into emp_master (empno,ename,salary) values (1122,
‘Smith’,8000);

1 row created.
```

Above command insert one row but values are inserted in only three columns. Remaining four columns have null values.

If you have define not null constraint in any of remaining columns it want allow you to insert data in a table.

Adding values in a table using Variable method

Till now we have seen static method to insert data. One can add data in a table using variable method with ***&*** (ampersand) sign. It will prompt user to enter data of mention field. Generally It is used to add more than one row in a table without typing whole command repetitively using / sign.

```
SQL>insert into emp_master values(&empno, ’&ename’, ’&job’,
‘&hiredate’, &salary, &deptno, &comm.);
```

When you terminate the command Oracle will prompt you to enter the values for the columns.

```
   Enter value for empno: 1123
   Enter value for ename: King
   Enter value for job: Clerk
   Enter value for hiredate: 30-jun-00
   old1: insert into emp values (&empno, ‘&ename’, ‘&job’ ,
   new1: insert into emp values(1124,’King’,’Clerk’,
   Enter value for sal: 3400
   Enter value for deptno: 20
   Enter value for comm: 300
   old2: ‘&hiredate’,&sal,&deptno,&comm)
   new2: ’30-jun-00',3400,20,300)

1 row created.

   SQL >/

   Enter value for empno: 1124
   Enter value for ename: Martin
   Enter value for job: Manager
   Enter value for hiredate: 30-aug-00
   old1: insert into emp values(&empno, ‘&ename’, ‘&job’,
   new1:insert into emp values(1124,’Martin’,’Manager’,
   Enter value for sal: 7000
   Enter value for deptno: 20
   Enter value for comm: 700
   old   2: ‘&hiredate’,&sal,&deptno,&comm)
   new   2: ’30-aug-00',7000,20,700)

1 row created.
```

If you want to put null value in a particular column then following command performed.

```
SQL>insert into emp values(1125,’Tanmay’,null,’16-sep-00’, null, 10,
null);

1 row created.
```

Simple Select Command

Stored information can be retrieved from the table through select command. Select is the most frequently used command, as access to information is needed all the time. Syntax of simple select command is as per follows:

`Select <column1>,<column2>,…,<column(n)> from  <table name>;`

The following command will select all the rows and columns from *emp_master*.

```
   SQL> select * from emp_master;

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  1122    Allen     Manager   1-JAN-00    10000    10      1000
  1122    Smith               1-JAN-00    8000
  1123    King      Clerk     30-JUN-00   3400     20      300
  1124    Martin    Manager   30-AUG-00   7000     20      1000
  1125    Tanmay              16-SEP-00            10

5 rows selected.
```

The ‘*’ will indicate all the columns. But If you want to retrieve only specific columns from a table then you have to specify column names with select commands.

```
   SQL> select empno,ename,salary from emp_master;
```

This query will give information from only three columns.

```
  EMPNO     ENAME         SALARY
  ——        ———           ———
  1122      Allen         10000
  1122      Smith         8000
  1123      King          3400
  1124      Martin        7000
```

```
5 rows selected.
```

Update command

Sometimes changes should be required in existing data. To reflect such changes to the existing records in a table the update command is used. We can update single column or more than one columns. Specific rows could be updated based on some condition. If condition is not specified with where clause it will update all the rows of specified column.

```
Update <table name> set <colum nname> = <value>  [where <condition>];
```

For example, if one want to change the salary of all the employee who are working in Department number 10.

Records before update command.

```
SQL> select * from emp_master;

  EMPNO   ENAME     JOB        HIREDATE    SALARY   DEPTNO  COMM

  1122    Allen     Manager   1-JAN-00     10000    10      1000
  1122    Smith               1-JAN-00     8000
  1123    King      Clerk     30-JUN-00    3400     20      300
  1124    Martin    Manager   30-AUG-00    7000     20      1000
  1125    Tanmay             16-SEP-00              10

5 rows selected.
```

Update command to change salary.

```
SQL>update emp_master set salary=10000 where deptno=10;

2 row updated.
```

After update command is performed the output will be as follows:

```
SQL> select * from emp_master;

  EMPNO   ENAME     JOB        HIREDATE    SALARY   DEPTNO  COMM

  1122    Allen     Manager    1-JAN-00    10000    10      1000
  1122    Smith                1-JAN-00    8000
  1123    King      Clerk      30-JUN-00   3400     20      300
  1124    Martin    Manager    30-AUG-00   7000     20      1000
  1125    Tanmay               16-SEP-00   10000    10

5 rows selected.
```

Example

```
SQL>update emp_master set COMM=500 where deptno=10;

5 rows updated.
```

After update command is performed the output will be as follows:

```
SQL> select * from emp_master;

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----

  1122    Allen     Manager   1-JAN-00    10000    10      500
  1122    Smith               1-JAN-00    8000     500
  1123    King      Clerk     30-JUN-00   3400     20      500
  1124    Martin    Manager   30-AUG-00   7000     20      500
  1125    Tanmay              16-SEP-00   10000    10      500

5 rows selected.
```

Delete command

To delete any inserted row, delete command is used. Delete command can also be used with where condition to delete only specific rows. If where condition is not specified then it will delete all the rows but structure remain as it is.

`Delete [from] <table_name> [where <condition>];`

Example

Delete rows from emp_master where deptno is null. Before delete command:

```
SQL> Delete from emp_master where deptno is null;

1 row deleted.
```

After delete command is performed. The output will be

```
SQL> select * from emp_master;

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----

  1122    Allen     Manager   1-JAN-00    10000    10      500
  1123    King      Clerk     30-JUN-00   3400     20      500
  1124    Martin    Manager   30-AUG-00   7000     20      500
  1125    Tanmay              16-SEP-00   10000    10      500

4 rows selected.
```

Select Command

Previously we have seen simple use of select statement to retrieve the data from the table. Now we have look further use of Select statement.

Distinct Clause

To prevent the selection of distinct rows, we can include distinct clause with select command.
The following command will exclude duplicate empno.

```
   SQL> select distinct deptno from emp_master;

   DEPTNO
   ——-
   10
   20

2 rows selected.
```

Select command with where clause:

To list out specific rows from a table we can include where clause. We have to specify conditions with where clause to filter the records. The where clause is similar which we have used with delete and update command. It can be done using :

```
   Select <column(s)> from <table name> where [condition(s)];
```

Example
Suppose you want to view only those rows where HireDate is 1-JAN-00.

```
   SQL> select empno,ename from emp_master where hiredate = ‘1-jan-
   00’;

   EMPNO     ENAME
   ——        ———-
   1122      Allen

1 row selected.
```

Order By Clause

Order by clause is used to arrange rows in either ascending or descending order. The order by clause can also be used to arrange multiple columns. The order by clause should be the last clause in select statement. It is used as per follows :

```
select  <column(s)> from <TableName> where [condition(s)]  [order by
<column name> [asc/]desc];
```

Example
If you want to view salary in ascending order the following command can performed:

```
SQL> select empno,ename,salary from emp_master order by salary;

   EMPNO   ENAME   SALARY
   __      ___     ___
   1123    King    3400
   1124    Martin  7000
   1122    Allen   10000
   1125    Tanmay  10000

4 rows selected.
```

If you have not specify any order by default it will consider ascending order and salary will be displayed in ascending order. To retrieve data in descending order the **desc** keyword is used after order by clause.

```
SQL> select empno,ename,salary from emp_master order by salary desc;
```

And the output will opposite from above.

```
   EMPNO ENAME     SALARY
   __ ___   ___
   1122    Allen   10000
   1125    Tanmay  10000
   1124    Martin  7000
   1123    King    3400

4 rows selected.
```

Select command with DDL and DML command.

Select command is used to provide information of the table. But apart from retrieving data it is used with some DDL and DML commands.

Table Creation with select statement

One can create table using select statement as per follows :

`create table <table name> as select <columnname(s)> from <existing table name>;`

Example
Using following command we can copy emp_master table into emp_master_copy.

```
SQL>create table emp_master_copy  as select * from emp_master;
```

It will create *emp_master_copy* table with the same Structure and data of *emp_master* table. Suppose if you want to create new table with some specific columns only, then you have to specify column name in select statement as per follows:

```
SQL>create table emp_master_copy1 (eno,nm) as select empno,ename from
emp_master;
```

This command will create a new table *emp_master_copy1* with only two columns *eno* and *nm* similar to *empno* and *ename* available in *emp_master*

If you want to make a copy of table without any data i.e. only structure of table, one have to specify wrong condition (like 1=2, 2=3,11=13).

```
SQL>create table emp_copy as select * from emp_master where 1=2;
```

The condition 1=2 will never satisfy so select statement will retrieve none row and only structure will copy.

Insert data using Select statement
Inserting records from one table to another table can also possible through select statement.

Syntax:

```
Inert into <tablename> (select <columns> from <tablename>);
```

Example

```
SQL> insert into emp_copy (select * from emp_master);
```

This will insert all the rows of emp_master.

> ☑ ***When you insert data into one table from another table data structure should be same of both the table.***

If you want to make copy of selected columns data from one table to another table the data structure of both the columns should be same.

Example

```
SQL> insert into emp_copy(nm) (select name from emp_master);
```

Change Table Name

One can change the existing table name with a new name.

Syntax

```
Rename <OldName> To <NewName>;
```

Example:
```
SQL> Rename emp_master_copy1 To emp_master1;

Table Renamed.
```

Transaction Control Language

All the changes made to the database can be referred to as a transaction. A transaction begins with DML commands but ends explicitly with transaction control commands.

Following commands categorized as Transaction control commands

- Commit
- Savepoint
- Rollback

Commit Command

This command is used to end a transaction. Only with the help of commit commands transactions changes can be made permanent to the database. This command also erases all the savepoints and releasing transaction locks.

Syntax:
```
     Commit Work;
          Or
     Commit;
```

Remember that all the DDL commands are auto committed but you have to commit all the DML commands explicitly.

Savepoint Command

Savepoint command is used to mark very lengthy transaction into smaller parts. They are used to identify a point in a transaction to which we can later rollback.

Syntax:
```
     Savepoint <savepoint_name>;
```

Example:
```
     SQL> Savepoint s1;
```

Rollback command

Rollback command undoes the work done in the current transaction from the starting or last commit. When we perform rollback it will rollback up to last commit or specified savepoint.

Syntax :
```
     Rollback Work;
```

```
          Or
     Rollback;
          Or
     Rollback to Savepoint <save_point>;
```

Nullifying effect to specific transaction.

```
SQL> Rollback to Savepoint s1;
```

It will rollback transaction up to Savepoint *s1*. It will rollbacked all the transaction from savepoint to last transaction.

Data Control Language

Data control language provides users with privilege commands. The owner of the database object has the authority over them.

Following commands are categorized as Data Control commands

- Grant command
- Revoke command

Grant Command

When you create any database object (till now you are aware with table) you are the owner of that object. The ball is in your golf that you want to share an object with other users or not. Grant command gives the facility so that you can share an object with privileges. You can restrict other user to perform certain operation on your object. Objects are logical storage structure like tables, views, sequences, indexes, synonyms etc.

Syntax:

```
     Grant <privileges> on <object name> to <user name>;
```

We can specify *privileges* like select, update, delete, insert. We can specify all to grant all privileges.

Example:

If you are admin user and if you want to give grant of your emp_master table to user scott with select and update option.

```
SQL>Grant select,update on emp_master to scott;
Grant succeeded.
```

The user scott will only perform select and update commands on emp_master table. The granted table can be used with **dot notatoion**.

```
SQL>select * from admin.emp_master ;
```

Here we have to specify user name that give grant on emp_master. We want to give grant in such a way, so that user can give grant to other user on granted object.

```
SQL> Grant all on emp_master to scott with grant option;
Grant succeeded.
```

After this command user *scott* can give grant of *emp_master* to other users. User *admin* is owner of *emp_master* table.

```
SQL> Grant all on admin.emp_master to Tanmay;
Grant succeeded.
```

We will discuss more on it in DBA Part.

Revoke command

Revoke command is used to withdraw the privileges that have been granted to a user using Grant Command.

Syntax

```
Revoke <privilege(s)> on <object name> from <user name>;
```

Example

```
SQL> Revoke select,update on emp_master from scott;

Revoke succeeded.
```

After revoke command perform user scott will not be enable to use emp_master.

We will discuss more on it in DBA Part.

Operators

SQL *Plus having following operators.

- Arithmetic Operators
- Comparison Operators
- Logical Operator

Arithmetic Operator

Arithmetic operators are used to perform calculations based on number values. The arithmetic operators are + (addition), - (subtraction), * (multiplication) and / (division). We can include them in sql command.

Example

```
SQL> select salary+comm from emp_master;

  Salary+comm
  —————-
  11000
   3700
   8000
  (Null)

4 rows selected.
```

Example:

```
SQL> select salary+comm net_sal from emp_master;

  NET_S
  ———-
  11000
  3700
  8000
  (Null)

4 rows selected.
```

In above query, it will give output of salary+comm and *net_sal* is column alias, which is used to change column heading. So the output will be displayed under the *net_sal* heading. If you calculate any number value with null value, it will always return **null** value.

In arithmetic operators * and / have equal higher precedence. And + and – have equal lower precedence. Check the following illustrates the precedence of operators.

```
SQL> Select 12*(salary+comm) annual_netsal from emp_master;

  ANNUAL_
  ———-
  132000
  44400
  96000
  (Null)

4 rows selected.
```

If the parenthesis is omitted then multiplication will be performed first followed by addition. We can change the order of evaluation by using parenthesis.

Comparison Operators:

Comparison operators are used in condition to compare one expression with other. The comparison operators are =, >, <, >=, <=, !=, `between`, `like` , `is null` and `in` operators.

`Between` operator is used to check between two values.

Example:

```
SQL> select * from emp_master where salary between 5000 and 8000;

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----
  1124    Martin    Manager   30-aug-00   7000     20      1000

1 row selected.
```

The above select statement will display only those rows where salary of employee is between 5000 and 8000.

IN Operator:

The `in` operator can be used to select rows that match one of the values in a list.

```
SQL>Select * from emp_master where deptno in(10,30);

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----
  1122    Allen     Manager   1-JAN-00    10000    10      1000
  1125    Tanmay              16-SEP-00   10000    10

2 rows selected.
```

The above query will retrieve only those rows where *deptno* is either in 10 or 30.

LIKE Operator:

`Like` operator is used to search character pattern, we need not know the exact character value. The `like` operator is used with special character **%** and **_** (underscore).

```
SQL> select * from emp_master where job like ‘M%’;

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----
  1122    Allen     Manager   1-jan-00    10000    10      1000
  1124    Martin    Manager   30-aug-00   7000     20      1000

2 rows selected.
```

The above select statement will display only those rows where job is starts with ‘M’ followed by any number of any characters. % sign is used to refer number of characters (it similar to * asterisk wildcard in DOS), while _ (underscore) is used to refer single character(it similar to ? question wildcard in DOS).

```
SQL>Select * from emp_master where job like ‘_lerk’;
```

```
  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----
  1123    King      Clerk     30-jun-00   3400     20      300
```

```
1 row selected.
```

In above query, it will display only those rows where job is start with any single character but ends with 'lerk'.

Logical Operators:

Logical operators are used to combine the results of two conditions to produce a single result. The logical operators are AND, NOT and OR.

AND Operator:

The Oracle engine will process all rows in a table and display the result only when all the conditions specified using the AND operator are satisfied.

```
SQL> select * from emp_master where salary > 5000 and comm < 750 ;

No rows selected.
```

The select statement will return only those rows where salary is greater than 5000 and comm is less than 750. If both the conditions are true then only it will retrieve rows.

OR Operator:

The Oracle engine will process all rows in a table and display the result only when any of the conditions specified using the OR operators are satisfied.

```
SQL>select * from emp_master where salary > 5000 or  comm < 750;

  EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
  -----   ------    -------   ---------   ------   ------  ----
  1122    Allen     Manager   1-jan-00    10000    10      1000
  1123    King      Clerk     30-jun-00   3400     20      300
  1124    Martin    Manager   30-aug-00   7000     20      1000
  1125    Tanmay              16-SEP-00   10000    10

4 rows selected.
```

This select statement will check either salary is greater than 5000 or comm is less than 750. ie it will return all the records either of any one condition returns true.

NOT Operator:

The Oracle engine will process all rows in a table and display the result only when none of the conditions specified using the NOT operator are satisfied.

```
SQL> select * from emp_master where not salary = 10000;

   EMPNO   ENAME     JOB       HIREDATE    SALARY   DEPTNO  COMM
   -----   ------    -------   ---------   ------   ------  ----

   1123    King      Clerk     30-jun-00   3400     20      300
   1124    Martin    Manager   30-aug-00   7000     20      1000

2 rows selected.
```

This select statement will return all the records where salary is NOT equal to 10000.

Pre-define Functions

Oracle functions serve the purpose of manipulating data items and returning a result. Functions are also capable of accepting user-supplied variables or constants and operations on them. Such variables and constants are called *arguments*.

Functions are classified into *Group* Functions and *Single Row* Functions (*Scalar* Functions).

Before we check single row function and group function, we will take a look on "Dual table"

The Oracle Table "Dual"

Dual is a small oracle worktable, which consists of only one row and one column, and contains the value x in that column. Besides arithmetic calculations, it also supports date retrieval and it's formatting.

```
SQL> select 2*2 from dual;

   2*2
   —
     4
```

Single Row Functions (Scalar Functions):

Functions that act on only one value at a time are called as Single Row Functions. A Single Row function returns one result for every row of a queried table or view.

Single Row functions can be further grouped together by the data type of their arguments and return values. Functions can be classified corresponding to different data types as :

● String Functions	:	Work for String Data type
● Numeric Functions	:	Work for number Data type
● Conversion Functions	:	Work for conversion of one data type to another
● Date Functions	:	Work for Date Data type

String Functions:

String functions accept string input and return either string or number values.

1) Initcap (Initial Capital): This String function is used to capitalize first character of the input string.

Syntax:

```
    initcap(string)
```

Example:

```
SQL> select initcap('azure') from dual;

  INITC
  ——-
  Azure
```

2) Lower: This String function will convert input string in to lower case.

Syntax:

```
    Lower(string)
```

Example:

```
SQL> select lower('AZURE') from dual;

  LOWER
  ——-
  azure
```

3) Upper: This string function will convert input string in to upper case.

Syntax:

```
    Upper(string)
```

Example:

```
SQL> select upper('azure') from dual;

  UPPER
  ——-
  AZURE
```

4) Ltrim (Left Trim): Ltrim function accepts two string parameters; it will fetch only those set of characters from the first string from the left side of the first string, and displays only those characters which are not present in second string. If same set of characters are not found in first string it will display whole string

Syntax:

```
    Ltrim(string,set)
```

7Example:

```
SQL>select ltrim('azuretech','azure') from dual;

   LTRI
   ——
   tech
```

5) Rtrim (Right Trim): Rtrim function accepts two string parameters; it will fetch only those characters from the first string, which is present in set of characters in second string from the right side of the first string.

Syntax:

```
     Rtrim(string,set)
```

Example:

```
SQL>select rtrim('azuretrim','trim') from dual;

   RTRIM
   ——-
   azure
```

6) Translate: This function is useful when you want to encrypt string. It will take first character from string1 and search the same character in string2 if that character is found than it replaces that character out of string3 on base of position of character found in string2. In below given example first character "a" of string1 is found at position no 2 in string2, so it will extract second character from string3. Same way second character "b" is found at position number 4 in string2, so it will extract fourth character from string3 and so on. If any character in string1 is not found in string2 then it is kept unchanged.

Syntax:

```
     Translate(string1, string2, string3)
```

Example:

```
SQL>select translate('abcde','xaybzcxdye','tanzmulrye') from dual;

TRANS
——-
azure
```

7) Replace: This function is useful when you want to search a specified string and replace it with particlar string form the string provided. For example, you want to search 'A' from the 'TACHNOLOGIAS' and replace it with 'E' to make it 'TECHNOLOGIES'. Replace function accepts three arguments first argument is, from which string you want to search, second argument is what you want to search from the first argument and third argument is replace string, value of second argument, if found will be replaced with value passed in third argument.

Syntax:
```
     Replace(string, searchstring, replacestring)
```

Example:
```
SQL> select replace('jack and jue','j','bl') from dual;

   REPLACE('JACKA
   ———————
   black and blue
```

8) Substr: Substring fetches out a piece of the `string` beginning at `start` and going for `count` characters, if count is not specified, the string is fetched from start and goes till end of the string.

Syntax:
```
     Substr(string, starts [, count])
```

Example:
```
SQL>select substr('azuretechnology',4,6) from dual;

   SUBSTR
   ------
   retech
```

9) Chr: Character function except character input and return either character or number values. The first among character function is chr. This returns the character value of given number within braces.

Syntax:
```
     Chr(number)
```

Example:
```
SQL>select chr(65) from dual;

   C
   -
   A
```

10) Lpad (Left Pad): This function takes three arguments. The first argument is character string, which has to be displayed with the left padding. Second is a number, which indicates total length of return value and third is the string with which left padding has to be done when required.

Syntax:
```
     Lpad(String,length,pattern)
```

Example:
```
Sql > select lpad('Welcome',15,'*') from dual;
```

```
   LPAD(‘WELCOME’,
   ————————-
   ********Welcome
```

11) Rpad (Right Pad): Rpad does exact opposite then Lpad function.

Syntax:
```
     Lpad(String,length,pattern)
```

Example:
```
SQL> select rpad(‘Welcome’,15,’*’) from dual;

   RPAD(‘WELCOME’,
   ————————-
   Welcome********
```

12) Length: When the length function is used in a query. It returns length of the input string.

Syntax:
```
     Length(string)
```

Example:
```
SQL>select length(‘auzre’) from dual;

   LENGTH(‘AUZRE’)
   ————————-
   5
```

13) Decode: Unlike the translate function which performs character-by-character replacement the decode function does a value-by-value replacement.

Syntax:
```
Select decode(column name,if,then,if,then…....) from <tablename>;
```

Example:
```
SQL> select deptno,decode(deptno,10, ‘Sales’, 20, ‘Purchase’, ‘Account’)
DNAME from emp_master;

   DEPTNO    DNAME
   ———       ————
   10        Sales
   20        Purchase
   20        Purchase
   10        Sales

4 rows selected.
```

14) Concatenation (||) Operator: This operator is used to merge two or more strings.

Syntax:
```
     Concat(string1,string2)
```

```
SQL> select concat(‘Azure’,’ Technology’) from dual;

  CONCAT(‘AZURE’,’
  ————————
  Azure Technology

SQL> select ‘ename is ‘||ename from emp_master;

  ‘ENAME IS’||ENAME
  —————————-
  ename is Allen
  ename is King
  ename is Martin
  ename is Tanmay

4 rows selected.
```

Numeric Functions:

1) Abs (Absolute): Abs() function always returns positive number.

Syntax:
```
     Abs(Negetive Number)
```

Example:
```
SQL> select Abs(-10) from dual;

  ABS(-10)
  ————
  10
```

2) Ceil: This function will return ceiling value of input number. i.e. if you enter 20.10 it will return 21 and if you enter 20.95 then also it will return 21. so if there is any decimal value it will add value by one and remove decimal value.

Syntax:
```
     Ceil (Number)
```

Example:

```
SQL>select Ceil (23.77) from dual;

  CEIL(23.77)
  ——————-
  24
```

3) Floor: This function does exactely opposite of the ceil function.

Syntax:

```
     Floor(Number)
```

Example:

```
SQL>select Floor(45.3) from dual;

  FLOOR(45.3)
  ——————-
  45
```

4) Power: This function will return power of raise value of given number.

Syntax:

```
     Power(Number, Raise)
```

Example:

```
SQL>Select power (5,2) from dual;

  POWER(5,2)
  ——————
  25
```

5) Mod: The function gives the remainder of a value divided by another value.

Syntax:

```
     Mod(Number, DivisionValue)
```

Example:

```
SQL>select Mod(10,3) from dual;

  MOD(10,3)
  —————-
  1
```

6) Sign: The sign function gives the sign of a value without it’s magnitude.

```
SQL>select sign(-45) from dual;

  SIGN(-45)
  ————-
  -1
SQL>Select sign(45) from dual;

  SIGN(45)
  ————
  1
```

Date Function:

1) Add_Months: The add_months data function returns a date after adding a specified data with the specified number of months. The format is **add_months(d,n)**, where **d** is the date and **n** represents the number of months.

Syntax:

```
     Add_Months(Date,no.of Months)
```

Example:

```
SQL> select Add_Months(sysdate,2) from dual;
```

This will add two months in system date.

```
  ADD_MONTH
  ————
  02-NOV-01
```

2) Last_day: Returns the last date of month specified with the function.

Syntax:

```
     Last_day(Date)
```

Example:

```
SQL> select sysdate, last_day(sysdate) from dual;

  SYSDATE     LAST_DAY
  ————        ————-
  02-SEP-01   30-SEP-01
```

3) Months_Between: Where Date1, Date2 are dates. The output will be a number. If Date1 is later than Date2, result is positive; if earlier, negative. If Date1 and Date2 are either the same days of the month or both last days of months, the result is always an integer; otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of Date1 and Date2.

Syntax:
```
     Months_Between(Date1,Date2)
```

Example:
```
SQL>select months_between(sysdate,’02-AUG-01’)  “Months” from dual;

  MONTHS
  ———
  4
```

4) Next_Day: Returns the date of the first weekday named by ‘char’ that is after the date named by ‘Date’. ‘Day’ must be the day of the week.

Syntax:
```
     Next_Day(Date,Day)
```

Example:
```
SQL>select next_day(sydate, ‘sunday’) “Next” from dual;
```

This will return date of next sunday.

```
  NEXT_DAY
  ————-
  09-SEP-00
```

5) Round: This function returns the date, which is rounded to the unit specified by the format.

Syntax:
```
     Round (Date, [fmt])
```

If format is not specified by default date will be rounded to the nearest day.

Example:
```
SQL>Select round(‘4-sep-01’,’day’) “Rounded” from dual;

  Rounded
  ————-
  02-SEP-01
```

The date formats are ‘month’ and ‘year’.
If rounded with ‘month’ format it will round with nearest month.
If rounded with ‘year’ format it will round with nearest year.

6) Trunc (Truncate): This function returns the date, which is truncated to the unit specified by the format.

Syntax:

```
     Trunc(Date,[fmt])
```

If format is not specified by default date will be truncated.

Example:

This will display first day of current week.

```
SQL>Select Trunc('4-sep-01','day') "Truncated" from dual;

   Truncated
   ————-
   02-SEP-01
```

The date formats are 'month' and 'year'.
If rounded with 'month' format it will display first day of the current month.
If rounded with 'year' format it will display first day of the current year.

Conversion Functions:

Conversion functions convert a value from one data type to another. The conversion functions are classified into the following:

- To_Number()
- To_Char()
- To_Date()

1) To_Number: The to_number function allows the conversion of string containing numbers into the number data type on which arithmetic operations can be performed.

Example:

```
SQL>Select to_number('50') from dual;

   TO_NUMBER('50')
   ————————-
   50
```

2) To_Char: To_char function converts a value of number data type to a value of char data type, using the optional format string. It accepts a number (no) and a numeric format (fmt) in which the number has to appear. If 'fmt' is omitted, 'no' is converted to a char exactly long enough to hold significant digits.

Syntax:

```
     To_char(no,[fmt])
```

Example:

```
SQL> select to_char(17145,'$099,999') "Char" from dual;
```

```
   Char
   ————
   $017,145
```

To_char converts a value of date datatype to character value. It accpets a date, as well as the format(fmt) in which the date has to appear. ‘fmt’ must be the date format. If ‘fmt’ is omitted, ‘date’ is converted to a character value in the default date format “dd-mon-yy”.

Syntax:

```
     To_char(Date,[fmt])
```

Example:

```
SQL>select to_char(hiredate, ‘month dd yyyy’) “HireDate” from emp_master
where salary = 10000;

   HireDate
   —————————-
   January 01 2000
   September 16 2000
```

3) To_Date: The format is to_date(char [,fmt]). This converts char or varchar datatype to date datatype. Format model, fmt specifies the form of character. Consider the following example which returns date for the string ‘January 27 2000’.

Syntax:

```
     To_date(char,[fmt])
```

Example:

```
SQL>select to_date(’27 January 2000’,’dd/mon/yy’) “Date” from dual;

   Date
   ————-
   27-JAN-00
```

Miscellaneous Functions:

The following are some of the miscellaneous functions supported by Oracle.

- Uid
- User
- Nvl
- Vsize

1) Uid (User Id): This function returns the integer values corresponding to the user currently logged in. The following example is illustrative.

Example:
```
SQL> select uid from user;

   UID
   —-
   320
```

2) User: This function returns the login's user name, which is in varchar2 datatype. Consider the following example.

Example:
```
SQL> select user from dual;

   USER
   ——-
   Scott
```

3) Nvl (Null Value): The *nvl* function is used in cases where we want to consider *Null* values as another value.

Syntax:
```
     Nvl(expression1, expression2)
```

If the expression1 is null then nvl will return expression2, and if expression1 is not null then it will return expression1.

Example:
```
SQL> Insert into emp_master (empno,ename,salary) values
          (1125,'Ward',null);

SQL>select nvl(salary,0) from emp_master;

   SALARY
   ———
   10000
   3400
   7000
   10000
   0
5 rows selected.
```

☑ ***Null values and zeroes are not equivalent. Null values are represented by blank and zeroes are represented by (0).***

4) Vsize function: This function returns no. of bytes in the expression. If expression is Null, it returns Null.

Syntax:

```
     Vsize(expression)
```

Example:

```
SQL> select vsize('azure') from dual;

  VSIZE
  ——
  5
```

Group Functions:

A group functions returns a result based on a group of rows. Some of these are just purely mathematical functions. The group functions supported by Oracle are summarized below:

1) Avg (Average): This function will return the average of values of the column specified in the argument of the column.

Example:

```
SQL> select avg(comm) from emp_master;

  AVG(COMM)
  ————-
  766.66667
```

2) Min (Minimum): The function will give the least of all values of the column present in the argument.

Example:

```
SQL>Select min(salary) from emp_master;

  MIN(SALARY)
  —————-
  3400
```

3) Max (Maximum): To perform an operation, which gives the maximum of a set of values the max, function can be made use of.

Example:

```
SQL>select max(salary) from emp_master;
```

This query will return the maximum value of the column specified as the argument.

```
  MAX(SALARY)
  —————-
  10000
```

4) Sum: The sum function can be used to obtain the sum of a range of values of a record set.

Example:
```
SQL>Select sum(comm) from emp_master;

   SUM(COMM)
   ————-
   2300
```

5) Count: This function is used to count number rows. It can take three different arguments, which mentioned below.

Syntax:
```
      Count(*)
      Count(column name)
      Count(distinct column name)
```

Count (*): This will count all the rows, including duplicates and nulls.

Example:
```
SQL>Select count(*) from emp_master;

   COUNT(*)
   ————
   4
```

Count (Column name) : It counts the number of values present in the column without including nulls.

Example:
```
SQL> select count(comm) from emp_master;

   COUNT(comm)
   —————-
   3
```

Count (distinct column name) : It is similar to count(column name) but eliminates duplicate values while counting.

Example:
```
SQL>Select count(distinct deptno) from emp_master;

   COUNT(DEPTNO)
   ——————-
   2
```

Group By Clause

Group by clause is used with group functions only. Normally group functions returns only one row. But group by clause will group on that column.

The group by clause tells Oracle to group rows based on distinct values for specified columns, i.e. it creates a data set, containing several sets of records grouped together based on a condition.
Select group function from table name group by column name

Example:

```
SQL>select  deptno,count(*) from emp_master  group by deptno;

  DEPTNO    COUNT(*)
  ———       ————
  10        2
  20        2
            1
```

Having Clause

The **having clause** is used to satisfy certain conditions on rows, retrieved by using group by clause. Having clause should be preceding by a group by clause. Having clause further filters the rows return by **group by** clause.

Example

```
SQL> select  deptno,count(*) from emp_master  group by deptno having
Deptno is not null;

  DEPTNO  COUNT(*)
  ———     ————
  10      2
  20      2
```

Using ROLLUP, GROUPING, and CUBE

How can you perform grouping operations, such as totals, within a single SQL statement rather than via SQL * Plus commands? As of Oracle8i, you can use the ROLLUP and CUBE functions to enhance the grouping actions performed within your queries. The following listing shows a LEDGER query for items bought in March:

```
Select Person,Action,Sum(Amount) from LEDGER
     Where ActionDate between
     To_Date('01-Mar-2001','DD-MON-YYYY') and
     To_Date('31-Mar-2001','DD-MON-YYYY') and
     Action in('BOUGHT')
     Group By Person,Action;

  PERSON          ACTION        SUM(AMOUNT)

  Abhay Shah      BOUGHT        .35
  Tanmay Panchal  BOUGHT        .4
  Brijesh Vyas    BOUGHT        3.94
  Kamlesh Soni    BOUGHT        .25
```

```
Hemant Pandya   BOUGHT        25
Ankur Shah      BOUGHT        1.18
Nimish Shah     BOUGHT        .2
Vivek           BOUGHT        1
Dhaval          BOUGHT        6
```

Instead of simply grouping by Person and Action, you can use the **ROLLUP** function to generate subtotals and totals. In the following example, the **group by** clause is modified to include a **ROLLUP** function call. Notice the additional rows generated at the end of the result set.

```
Select Person,Action,Sum(Amount) from LEDGER
     Where ActionDate between
     To_Date('01-Mar-2001','DD-MON-YYYY') and
     To_Date('31-Mar-2001','DD-MON-YYYY') and
     Action in('BOUGHT')
     Group By Rollup(Action,Person);
```

```
PERSON          ACTION        SUM(AMOUNT)

Abhay Shah      BOUGHT        .35
Tanmay Panchal  BOUGHT        .4
Brijesh Vyas    BOUGHT        3.94
Kamlesh Soni    BOUGHT        .25
Hemant Pandya   BOUGHT        25
Ankur Shah      BOUGHT        1.18
Nimish Shah     BOUGHT        .2
Vivek           BOUGHT        1
Dhaval          BOUGHT        6
                BOUGHT        38.32
                              38.32
```

The two lines at the end of the listing are the rollup:- first by Action(the sum of all Bought amounts) and then for all Actions. Rather than using **break on action on report** command, the **ROLLUP** function within the **group by** clause generated the subtotals and totals within one query.

Let's refine the appearance of the report. The subtotal and total rows have a blank value listed under the columns by which the rollup is being performed: the subtotal line for the Action 'BOUGHT' has a blank entry for the Person; the total line for all Actions and Persons is blank for both of those columns. You can use the **GROUPING** function to determine whether the row is a total or subtotal (generated by **ROLLUP**) or corresponds to a **NULL** value in the database. In the **select** clause, the Person column will be selected as follows:

```
Select DECODE(GROUPING(Person),1,'All Persons',Person)
```

The **GROUPING** function will return a value of 1 if the column's value is generating by **ROLLUP** action. This query uses **DECODE** to evaluate the result of the **GROUPING** function. If the **GROUPING** output is 1, the value was generated by the **ROLLUP** function, and Oracle will print all persons; otherwise, it will print the value of the Person column. You can apply similar logic to the Action column. The full query is shown in the following listing along with its output:

```
Select DECODE(GROUPING(PERSON),1,’All Persons’,Person),
        DECODE(GROUPING(ACTION),1,’All Actions’,Action),
     Sum(Amount) from LEDGER
     Where ActionDate Between
     To_Date(’01-Mar-2001’,’DD-MON-YYYY’) and
     To_Date(’31-Mar-2001’,’DD-MON-YYYY’) and
     Action in(‘BOUGHT’)
     Group By ROLLUP(Action,Person);
```

```
  PERSON          ACTION        SUM(AMOUNT)

  Abhay Shah      BOUGHT        .35
  Tanmay Panchal  BOUGHT        .4
  Brijesh Vyas    BOUGHT        3.94
  Kamlesh Soni    BOUGHT        .25
  Hemant Pandya   BOUGHT        25
  Ankur Shah      BOUGHT        1.18
  Nimish Shah     BOUGHT        .2
  Vivek           BOUGHT        1
  Dhaval          BOUGHT        6
  All Persons     BOUGHT        38.32
  All Persons     All Actions   38.32
```

You can use the **CUBE** function to generate subtotals for all combinations of the values in the **group by** clause. For two values in the **ROLLUP** clause (as in the last example), two levels of subtotals will be generated. For two values in a **CUBE** clause, four values will be generated. Let’s expand the query of LODGING and WORKER to incorporate the WORKERSKILL table, and then determine which skill are to be found among the workers in each lodging. The following query uses the **CUBE** function to generate this information; a **break** command is added for readability:

```
Break on lodging due skip 1

Select DECODE(GROUPING(LODGING.Lodging),1,’All Lodgings’,
     LODGING.Lodging) As Lodging,
     DECODE(GROUPING(WORKERSKILL.Skill),1,’All Skills’,
     WORKERSKILL.Skill) As Skill,
     Count(*) As Total_Workers
     From WORLKER,LODGING,WORKERSKILL
     Where WORKER.Lodging=LODGING.Lodging And
     WORKER.Name=WORKERSKILL.Name
     Group By CUBE(LODGING.Lodging,Workerskill.Skill);
```

This query’s output is shown in the following listing:

```
  LODGING         SKILL           TOTAL WORKERS

  MULLERS         SMITHY                1
  MULLERS         All Skills            1
```

```
PAPA KING       WORK                 1
PAPA KING       All Skills           1

ROSE HILL       COMBINE DRIVER       1
ROSE HILL       SMITHY               2
ROSE HILL       WOODCUTTER           1
ROSE HILL       All Skills           4

WEITBROCHT      DISCUS               1
WEITBROCHT      All Skills           1

All Lodging     COMBINE DRIVER       1
All Lodging     DISCUS               1
All Lodging     SMITHY               3
All Lodging     WOODCUTTER           1
All Lodging     WORK                 1
All Lodging     All Skills           7
```

Since there are only then rows in WORKERSKILL, there aren't matches for all of the workers, However, this output shows Talbot where to find a combine driver and woodcutter – and the cube results also show the skills for which he may need to add more workers. If you had used **ROLLUP** in place of **CUBE,** Oracle would have displayed the All Lodgings – All skills row shown in the preceding listing, but not the five All Lodgings rows that precede it.

Set Operators

Set operators combine the results of two queries into a single one. The following set operators are available in SQL.

- Union
- Union All
- Intersect
- Minus

While we are using set operators the following points must be keep in mind

The queries, which are related by a set operator should have the same number of columns and the corresponding columns, must be of the data types.
Such a query should not contain any columns of long data type.
The label under which the rows are displayed are those from the first select statement.

Union: The union operator returns all distinct rows selected by two or more queries. The following example combines the result of two queries with the union operator, which eliminates duplicate rows.

```
SQL> select order_no from order_master;

  ORDER_NO
  ————
```

```
   O001
   O002
   O003
   O004

SQL> select order_no from order_detail;

   ORDER_NO
   ————
   O003
   O004
   O005
   O006
   O007
```

Now we check output using union operator.

Example:

```
SQL>select order_no from order_master union select order_no from
order_detail;

   ORDER_NO
   ————
   O001
   O002
   O003
   O004
   O005
   O006
   O007
```

Union All : The union all operators returns all rows selected by either query including duplicates. The following example combines the result with the aid of union all operator, which does not eliminates duplicate rows.

Example:

```
SQL> select order_no from order_master union all select order_no from
order_detail

   ORDER_NO
   ————
   O001
   O002
   O003
   O004
   O003
```

```
  O004
  O005
  O006
  O007
```

Intersect : The intersect operator outputs only rows produced by both the queries intersected i.e. the output in an intersect clause will include only those rows that are retrieved by both the queries.

Example:

```
SQL> select order_no from order_master intersect select order_no from
order_detail;

  ORDER_NO
  ————
  O003
  O004
```

Minus : The Minus operator outputs the rows produced by the first query, after filtering the rows retrieved by the second query.

Example:

```
SQL> select order_no from order_master minus select order_no from
order_detail;
  ORDER_NO
  ————
  O001
  O002
```

Joins

Sometimes we require to treat multiple tables as though they were a single entity. Then a single SQL sentence can manipulate data from all the tables. To achieve this, we have to join tables. The purpose of join is to combine the data spread across tables. A join is actually performed by the 'where' clause which combines the specified rows of tables.

Syntax for joining tables

```
select columns from table1, table2, ... where logical expression;
```

Basically there are three different types of joins:

- Simple Join
- Self Join
- Outer Join

Simple Join : This is the most frequently used join. It retrieves the rows from two tables having a common column and is further classified into equi-join and non-equi join. **Equi join** is based on equality and where clause uses comparison operator equal to (=) to perform a join. **Non-equi join** specifies the relationship between columns belonging to different tables by making use of relational operators(>,<, >=,<=, <>).

Example:
```
SQL> select * from order_master , order_detail where
           Order_master.order_no = order_detail.order_no;
```

This select statement will join the output of order_master and order_detail and display only those rows where order_master's order_no equals to order_detail's order_no. In the example the column name is prefixed by the table name because both the tables have the same column name i.e. order_no. Therefore to distinguish between them we use table names. If the column names are unique, then we need not prefix it with the table name.

Example:
```
SQL> select a.*, b.* from itemfile a, order_detail b where a.max_level
< b.qty_ord
and a.itemcode = b.itemcode;
```

This select statement will retrieve rows from itemfile and order_detail where qty_ord of order_detail table is less than max_level of itemfile and Itemcode are common in both the table. Here a and b is indicating table aliases name. To prevent ambiguity in a query we include table names in the select statements. Table aliases are used to make multiple table queries shorter and more readable.

Self Join : In some situations, you may find it necessary to join a table to itself, as though you were joining two separate tables. This is referred to as a self-join. In a self-join, two rows from the same table combine to form a result row. To join a table to itself, two copies of the very same table have to be opened in memory. Hence in the from clause, the table name needs to be mentioned twice. Since the table names are same, the second table will overwrite the first table and in effect, result in only one table being in memory. This is because a table name is translated into specific memory location. To avoid this, each table to be opened under an alias. Now these table aliases will cause two identical tables to be opened in different memory locations. This will result in two identical tables to be physically present in the computer's memory.

Display employees'salary with their manager's salary. (Use default emp table)

Example:
```
SQL>  select a.ename, a.salary, b.ename, b.salary from emp a, emp b
where a.mgr = b.empno;
```

This query will return employee name,salary with his manager's name and salary.

Outer Join : Outer join extends the result of simple join. An outer join returns all the rows returned by simple join as well as those rows from one table that do not match any row from the other table. This cannot be with a simple join. The outer join is represented by (+) sign.

Example:

```
SQL> select * from order_master a, order_detail b where
         a.order_no = b.order_no(+);
```

This select statement will return all the records from order_master and only matching records from order_detail. If (+) is not specified then it is simple join and it will retrieve only matching records from both tables.

Subquery

A subquery is a form of an SQL statement that appears inside another SQL statement. It is also termed as nested query. The statement containing a subquery is called parent query statement. The parent statement uses the rows returned by the subquery. Subquery is always enclosed within parenthesis. Subquery will be evaluated first followed by the main query.

Example:

```
SQL> select * from order_master where order_no = (select order_no from
order_detail where order_no = ‘O001’);
```

In this case subquery will execute first and the main query’s condition will work on subquery’s output. Now check the following select statement, what will the output.

Example:

```
SQL> select * from order_master where order_no = (select order_no from
order_detail);
```

It will return an error, ‘single-row subquery returns more than one row’.
When subquery returns more than one row we have to use operators like any, all, in or not in.

‘=any’ is equivalent to ‘in’ operator and ‘!=all’ is equivalent to not in.

Example:

```
SQL>Select * from order_master where order_no = any(select order_no
from  order_detail);

SQL> select * from order_master where order_no in(select order_no from
order_detail);
```

Both select statements are equal. After using ‘any’ operator it display records from order_master where any order_no is equal to order_no from order_detail.

Example:

```
SQL> select * from order_detail where qty_ord  =all(select qty_hand
from itemfile where itemrate =250);
```

In above example the subquery will display area that holds the itemrate that is less than 250. The main query will display details about orders only if qty_ord is greater than all the values return by the subquery.

Partition

Using Partitioned Tables

As of Oracle8, you can divide the rows of a single table into multiple parts. Dividing a table's data in this manner is called partitioning the table that is partitioned is called a partitioned table, and the parts are called partitions.

Partitioning is useful for very large tables. By splitting a large table's rows across multiple smaller partitions, you accomplish several important goals:

The performance of queries against the tables may improve, since Oracle may have to only search on partition (one part of the table) instead of the entire table to resolve a query.
The table may be easier to manage. Since the partitioned table's data is stored in multiple parts, it may be easier to load and delete data in the partitions than in the large table.
Backup and recovery operations may perform better. Since the partitions are smaller than the partitioned table, you may have more options for backing up and recovering the partitions than you would have for a single large table.

The Oracle optimizer will know that the table has been partitioned; as shown later in this section, you can also specify the partition to use as part of the form clause of your queries.

Creating a Partitioned Table

To create a partitioned table, you specify how to set up the partitions of the table's data as part of the create table command. Typically, tables are partitioned by ranges of values.

Consider the EMPLOYEE table:

```
SQL>Create Table Employee
(EmpNo Number(3) Primary Key,Ename Varchar2(20),
 Salary Number(8,2),Deptno Number(3));

Table created.
```

If you will be storing a large number of records in the EMPLOYEE table, then you may wish to separate the EMPLOYEE rows across multiple partitions. To partition the table's records, use the **partition by range** clause of the **create table** command, as shown next. The ranges will determine the values stored in each partition.

```
SQL>Create Table Employee
(EmpNo Number(3) Primary Key,Ename Varchar2(20),
 Salary Number(8,2),Deptno Number(3))
 Partition by range(Salary)
(Partition Part1 values less than(2001),
 Partition Part2 values less than(4001),
 Partition Part3 values less than(6001),
 Partition Part4 values less than(MAXVALUE));
Table created.
```

The **EMPLOYEE** table will be partitioned based on the values in the **Salary** column.

```
    Partition by range(Salary)
```

For any salary values less than 2001, the record will be stored in the partition named **Part1.** Any salary in the range between 2001 and 4000 will be stored in the **Part2** partition; values between 4001 and 6000 will be stored in the **Part3** partition. Any value greater than 6000 will be stored in **Part4** partition definition, the range clause is

```
    Partition Part4 values less than(MAXVALUE)
```

You do not need to specify a maximum value for the last partition; the **MAXVALUE** keyword tells Oracle to use the partition to store any data that could not be stored in the earlier partitions.

Range partitions were the only type of partition available in Oracle8. **Oracle8i** also includes **hash partitions**. A hash partition determines the physical placement of data by performing a hash function on the values of the partition key. In range partitioning, consecutive values of the partition key are usually stored in the same partition. In hash partitioning, consecutive values of the partition key are not necessarily stored in the same partition. Hash partitioning does, potentially decreasing the likelihood for I/O contention.

To create a hash partition, use the **partition by hash** clause in place of the **partition by range** clause, as shown in the following listing:

```
SQL>Create Table Employee(
    (EmpNo Number(3) Primary Key,Ename Varchar2(20),
     Salary Number(8,2),Deptno Number(3))
     Partition By Hash(Salary)
     Partitions 10;

Table created.
```

You can name each partition and specify its tablespace, just as you would for range partitioning, as shown here:

```
SQL>Create Table Employee(
    (EmpNo Number(3) Primary Key,Ename Varchar2(20),
     Salary Number(8,2),Deptno Number(3))
     Partition By Hash(Salary)
     Partitions 2
     Store in(PART1_TS,PART2_TS);

Table created.
```

Following the **Partition By Hash(Salary)** line, you have two choices for format:

As shown in the preceding listing, you can specify the number of partitions and the tablespaces to use:

```
       Partitions 2
        Store in(PART1_TS,PART2_TS);
```

This method will create partitions with system-generated names of the format *SYS_Pnnn*. The number of tablespaces specified in the store in clause does not have to equal the number of partitions. If more partitions than tablespaces are specified, the partitions will be assigned to the tablespaces in a round-robin fashion.

You can specify named partitions:

```
     Partition By Hash(Salary)
(Partition P1 tablespace P1_TS,
      Partition P2 tablespace P2_TS);
```

In this method, each partition is given a name and a tablespace, with the option of using an additional **lob** or **varray** storage clause. This method gives you more control over the location of the partitions, with the added benefit of letting you specify meaningful names for the partitions.

Creating Subpartitions

As of Oracle8i, you can create subpartitions – partitions of partitions. You can use subpartitions to combine the two types of partitions: **range** partitions and **hash** partitions. You can use hash partitions in combination with range partitions, creating hash partitions of the range partitions. For very large tables, this composite partitioning may be an effective way of separating the data into manageable and tunable divisions.

The following example range-partitions the EMPLOYEE table by the salary column, and hash-partitions by ename values: -

```
SQL>Create Table Employee(
     (EmpNo Number(3) Primary Key,Ename Varchar2(20),
      Salary Number(8,2),Deptno Number(3))
      Partition By Range(Salary)
      Subpartition by Hash(Ename)
      Subpartitions 10
     (Partition PART1 values less than(2001)
      tablespace PART1_TS,
      Partition PART2 values less than(4001)
      tablespace PART2_TS,
      Partition PART3 values less than(6001)
      tablespace PART3_TS,
      Partition PART4 values less than(MAXVALUE)
      tablespace PART4_TS);

Table created.
```

The EMPLOYEE table will be range-partitioned into four partitions, using the ranges specified for the four named partitions. Each of those partitions will be hash-partitioned on the Ename column.

Indexing Partitions

When you create a partitioned table, you should create an index on the table. The index may be partitioned according to the same range values that were used to partition the table. In the following listing, the **create index** command for the EMPLOYEE table is shown:

Example:

```
SQL>Create Index Employee_Salary
     On Employee(Salary)
     local
     (Partition PART1
      tablespace PART1_NDX_TS,
      Partition PART2
      tablespace PART2_NDX_TS,
      Partition PART3
      tablespace PART3_NDX_TS,
      Partition PART4
      tablespace PART4_NDX_TS);
Index created.
```

Notice the **local** keyword. In this **create index** command, no ranges are specified. Instead, the **local** keyword tells Oracle to create a separate index for each partition of the EMPLOYEE table. There were four partitions created on EMPLOYEE. This index will create four separate indexes – one for each partition. Since there is one index per partition, the indexes are **"local"** to the partitions.

You can also create **"global"** indexes. A global index may contain values from multiple partitions. The index itself may be partitioned, as shown in this example:

Example:

```
SQL>Create Index Employee_Salary
     On Employee(Salary)
     global Partition by range(Salary)
     (Partition PART1 values less than(2001)
      tablespace PART1_NDX_TS,
      Partition PART2 values less than(4001)
      tablespace PART2_NDX_TS,
      Partition PART3 values less than(6001)
      tablespace PART3_NDX_TS,
      Partition PART4 values less than(MAXVALUE)
      tablespace PART4_NDX_TS);
Index created.
```

The **global** clause in this **create index** command allows you to specify ranges for the index values that are different from the ranges for the table partitions. Local indexes may be easier to manage than global indexes; however, global indexes may perform uniqueness checks faster than local (partitioned) indexes perform them.

Note: You cannot create global indexes for hash partition or subpartitions.

Managing Partitioned Table

You can use the **alter table** command to **modify, add, split, rename, truncate, drop, exchange** and **move** partitions. These **alter table** command option allow you to alter the existing partition structure, as may be required after a partitioned table have been use heavily. For example, the distribution of Salary values within the partitioned table may have changed, or the maximum value may have increased.

To Modify Existing Partition Range:

```
SQL>Alter Table Employee Modify Partition P4 values less than(8001);

Table altered.
```

To Add New Partition:

```
SQL>Alter Table Employee Add Partition P5 values less than(MAXVALUE);
```

(To add single partition, you can use above command.)

Or

```
SQL>Alter Table Employee Add(Partition P5 values less than(10001),Partition P6 values less than(MAXVALUE));
```

(To add multiple partition, you can use above command.)

```
Table altered.
```

Note: Table can have must be one partition otherwise you cannot add new partition in table.

To Split Partition:

The split partition clause can be used to split a partition into two. This is useful when a partition becomes too large and causes backup, recovery or maintenance operations to consume a lot of long time.

```
SQL>Alter Table Employee split partition P5 at(10001);
```

(If you can use above command then Oracle will give name of both partitions.)

Or

```
SQL>Alter Table Employee split partition P5 at(10001) into(Partition P51,Partition P52);
```

(If you want to give partitions name then use above command.)
```
Table altered.
```

To Rename Existing Partition:

```
SQL>Alter Table Employee Rename Partition P51 To P5;
Table altered.
```

To Truncate Partition:

```
SQL>Alter Table Employee Truncate Partition P52;
Table altered.
```

This command will delete all the records of partition p52. But partition is available in the table. You cannot rollback this transaction.

To Drop Partition:

```
SQL>Alter Table Employee Drop Partition P52;
Table altered.
```

This command will delete partition p52 with all the record.

To Exchange Partition:

Exchanging table partitions is used convert a partition into a non-partitioned table, and a table into a partition of a partitioned table by exchanging their data segments. Exchanging table partitions is most useful when an application using non-partitioned tables need to be converted into partitions of a partitioned table.

```
SQL>Alter Table Employee Exchange Partition P2 with table EmpDet;
Table altered.
```

Note: Table **EmpDet** can have those records, which are fulfill condition of **P2** partition of **Employee** table.

To Move Partition:
The Move Partitions clause of the Alter Table statement is used to move a partition from a most active tablespace to a different tablespace in order to balance I/O Operations.

```
SQL>Alter Table Employee Move Partition P3 Tablespace ROLLBACK_DATA;
Table altered.
```

Locks

Concept of Locking

Users manipulate Oracle table data via SQL or PL/SQL. An Oracle transaction can be made up of a Single Query Transaction (SQT) or Multiple Query Transaction (MQT). These transactions access an

Oracle table or tables. Oracle works on multi-user platform, it is more than likely that several people will access data either for viewing or for manipulating from the same tables at the same time via different SQL statements. The Oracle table is therefore a global resources (shared by several users).

There is definite need to ensure the integrity of data in a table is maintained each time that its data is accessed. The Oracle engine has to allow simultaneous access to table data without causing damage to the data. The technique employed by Oracle engine to protect table data when several people are accessing it is called **Concurrency Control**. Oracle uses a method called Locking to implement concurrency control when multiple users access a table to manipulate its data at the same time.

Locks

Locks are mechanisms used to ensure data integrity while allowing maximum concurrent access to data. Oracle's locking is **fully automatic** and **requires no user intervention**. The Oracle engine automatically locks the table data while executing SQL statements. This type of locking is called implicit locking.

Since the Oracle engine has a fully automatic locking strategy, it has to decide on following two :

- Level of Lock to be applied
- Type of Lock to be applied

Levels of Locks

A table can a decomposed into rows and row further decomposed into columns. Oracle does not provide column level lock. We can lock on either row level, page level or table level.
The Oracle engine decides on the level to be used by the presence or absence of a where clause conditions in the SQL statements.

- If the where condition evaluates to only one row in the table, a row level lock is used.
- If the where condition evaluates to set of rows in the table, a page level lock is used.
- If the query accesses the entire table, a table level lock is used.

The Select ... For Update statement

It is used for acquiring exclusive row level locks in anticipation of performing updates on records. This clause generally used to signal the Oracle engines that data currently being used need to be updated. It is often followed by one or more update statements with a where clause.

Example:
```
SQL> select * from emp_master where empno = 1122 for update;
```

Other users can performs manipulations other than update statement of the row that has been lock is released.

Nowait clause

In order to avoid unnecessary waiting time, a nowait option can be used to inform the Oracle engine to terminate the SQL statement if the record has already locked. If this happens the Oracle engine terminates the running DML statement and comes up with a message indicating that the resource is busy.

Example:

```
SQL> select * from emp_master where salary > 5000 for update nowait;
SQL>Lock table emp_master in exclusive mode nowait;
```

Tables can be locked in several modes, share lock, share update, exclusive lock.

Syntax:

```
Lock table <Table Name> in <share or share update or exclusive mode>;
```

There are three types of locks supported by Oracle are:

1) Share Lock : Shared lock locks the table allowing users to only query but not insert, update, or delete rows from a table. Multiple shared lock can be simultaneously set on a resource. It allows resources can be shared, hence the name is share lock.

Example:

```
SQL> lock table emp_master in share mode;
```

This command will lock order_master in share lock.

2) Share Update Lock : It locks the rows that are to be updated in a table. It allows other users to concurrently query, insert, update or even lock other rows in the same table. It prevents the other users fro updating the row that has been locked.

Example:

```
SQL> lock emp_master in share update mode;
```

3) Exclusive Lock : Exclusive locks are the most restrictive of table locks. When user lock tables in exclusive mode other user can only query but insert, update, delete is not allowed. Only difference between share lock and exclusive lock is, only one user can issue exclusive lock on a table at a time. When lock is release then only other user can issue exclusive lock.

Example:

```
SQL> lock table emp_master in exclusive lock;
```

Releasing Locks

Locks are released under following circumstances:

- The transaction is committed successfully using the “Commit” command.
- A rollback is performed.
- A rollback to a savepoint will release locks set after the specified savepoint.

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 catalogue. 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, very large extent. When a View is used to manipulate table data, the underlying base table will be 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 that a query fired on a base table. This is because the View definition has to be retrieved from Oracle's system catalogue, 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 View's are used only for looking at table data. Other View's 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 securiry

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

Syntax:
```
     Create View <View_Name> As Select statement;
```

Example:
```
SQL>Create View EmpView As Select * from Employee;
View created.
```

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

Syntax:
```
     Select columnname,columnname from <View_Name>;
```

Example:
```
SQL>Select Empno,Ename,Salary from EmpView where
     Deptno in(10,30);

  EMPNO     ENAME       SALARY
  ——        ———         ———
  1122      Allen       10000
  1125      Tanmay      10000

2 rows selected.
```

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 you give an updateable view name in the Update, Insert or Delete SQL statement, modifications to data will be 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:
Table Name: Employee

Column Name	Data Type	Size	Attributes
Empno	Number	3	Primary Key
Ename	Varchar2	30	Not Null
Salary	Number	8,2	Not Null
Deptno	Number	3	

Syntax for creating an Updateable View:

```
Create View Emp_vw As
     Select Empno,Ename,Deptno from Employee;
View created.
```

→ When an **INSERT** operation is performed using the view:

```
SQL>Insert into Emp_vw values(1126,'Brijesh',20);
1 row created.
```

→ When an **MODIFY** operation is performed using the view:

```
SQL>Update Emp_vw set Deptno=30 where Empno=1125;
1 row updated.
```

→ When an **DELETE** operation is performed using the view:

```
SQL>Delete from Emp_vw where Empno=1122;
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 condition specified in the where clause of the View's 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.

View 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-preserved
table.
```

For delete:

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

View 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:
Table Name: Employee

Column Name	Data Type	Size	Attributes
Empno	Number	3	Primary Key
Ename	Varchar2	30	Not Null
Salary	Number	8,2	Not Null
Deptno	Number	3	Foreign Key references Deptno of DeptDet table

Table Name: DeptDet

Column Name	Data Type	Size	Attributes
Deptno	Number	3	Primary Key
Dname	Varchar2	30	Not Null

Syntax for creating a Master/Detail View (Join View):

```
SQL>Create View EmpDept_Vw As
        Select a.Empno,a.Ename,a.Salary,a.Deptno,b.Dname
        From Employee a,DeptDet b
        Where a.Deptno=b.Deptno;
View created.
```

When an **INSERT** operation is performed using the view

```
SQL>Insert into EmpDept_Vw values(1127,'Abhay
Shah',10000,10,'Technical');

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

When an **MODIFY** operation is performed using the view

```
SQL>Update EmpDept_Vw set salary=4300 where Empno=1125;
1 row updated.
```

When an **DELETE** operation is performed using the view

```
SQL>Delete From EmpDept_Vw where Empno=1123;
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 Salary * 2.25.
- 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-01776:data manipulation operation not legal on this view
```

To Create Read-only View:
In this view, you cannot manipulate the records. Because of this view is created with read only.

```
SQL>Create View EmpRO As select * from Employee with Read Only;
View created.
```

To Create View With Check option:
In this view, you cannot change value of deptno column. Because of this view is created with check option.

```
SQL>Create View EmpCk As Select * from Employee Where Deptno=10 With
Check Option;
View created.
```

Destroying a view:

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

Syntax:

```
     Drop View <View_Name>;
```

Example:
Remove the view Emp_Vw from the database.

```
SQL>Drop View Emp_Vw;
```

```
View dropped.
```

Synonyms

Synonym, it is called second name and short name of the table. You can use Synonym Name as the place of Table Name. You can operate table-using synonym.

Syntax:

```
Create Synonym <Synonym_name> for <Table Name>;
```

Example:

```
SQL>Create Synonym Emp for Employee;
Synonym created.
```

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 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 data entry is done by human beings it is quite likely that duplicate value will be entered, which violates the constraint and the entire row is rejected.

If the value to be entered into this column is machine generated, it will always fulfill the 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
provide intervals between numbers
caching of sequence numbers in memory etc.

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. The SQL statement used for creating a sequence is:

Syntax:

```
     Create Sequence <Sequence_name>
          [Increment By integervalue
           Start With integervalue
```

```
Maxvalue integervalue/Nomaxvalue
Minvalue integervalue/Nominvalue
Cycle/NoCycle
Cache integervalue/NoCache
Order/NoOrder]
```

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. By default, sequence is nocycle.

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. By default, cache of sequence is 20.

ORDER:
This guarantees that sequence numbers are generated in the order of request. This is only necessary if you are 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 only 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.

Example:
Create a sequence by the name Order_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 9999.

```
SQL>Create Sequence Order_seq increment by 1 start with 1 minvalue 1
maxvalue 9999 cycle;

Sequence created.
```

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:

```
SQL>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:
Insert values for Order_no, Order_date, Client_no in the Sales_order table. The **Order_seq** sequence must be used to generate Order_no and Order_date must be set to system date.

Table Name: Sales_Order

Column Name	**Data Type**	**Size**	**Attributes**
Order_no	Varchar2	6	Primary Key
Order_date	Date		
Client_no	Varchar2	6	
Dely_addr	Varchar2	25	
Salesman_no	Varchar2	6	
Dely_type	Char	1	Delivery : Part(P)/ Full(F) Default: ‘F’
Billed_yn	Char	1	
Dely_date	Date		
Order_status	Varchar2	10	

Syntax:
```
SQL>Insert Into Sales_order(Order_no,Order_date,Client_no)
values(Order_seq.Nextval,sysdate,’C00001’);
```

To reference the current value of a sequence:

```
SQL>Select SequenceName.Currval from Dual;
```

This is how, 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 Order_no stored in the Sales_order table, can be a concatenation of the month and year from the system date and the number generated by the sequence Order_seq. For example Order_no 01981 is generated with 01 (month in number format), 98 (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 sales orders for the month of January 98, the Order_no will start with 01981 upto 019850. Again when the month changes to February and as the sequence is reset, the numbering will start with 02981, 02981……

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

Example:

```
SQL>Insert Into Sales_order(Order_no,Order_date,client_no)
Values(to_char(sysdate,'MMYY'||to_char(Order_seq.nextval),
sysdate,'C00001');
1 row created.
```

Altering Sequence:

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

Syntax:

```
Alter Sequence <Sequence_name>
          [Increment By integervalue
           Maxvalue integervalue/NoMaxvalue
           Minvalue integervalue/NoMinvalue
           Cycle/NoCycle
           Cache integervalue/NoCache
           Order/NoOrder]
```

Example:
Changing Cache value of the sequence order_seq to 30 and interval between two numbers as 2.

```
SQL>Alter Sequence Order_seq Increment By 2 Cache 30;
Sequence altered.
```

Dropping Sequence:

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

Syntax:
```
    Drop Sequence <Sequence_Name>;
```

Example:
Remove the sequence Order_seq.

```
    SQL>Drop Sequence Order_seq;
```

Indexes

When the user fires a SELECT statement 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 locates the starting 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.

For example, to locate all the orders placed by client 'C00001' held in the sales_order table the Oracle engine must first locate the sales_order table and then perform a sequential search on the client_no column seeking a value equal too 'C00001'.

The records in the sales_order table are stored in the order in which they are keyed in and thus to get all orders where client_no is equal to 'C00001' 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 the record/s can be located and retrieved from a table.

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.

A column, which will hold sorted data, extracted from the table on which the index is being created.
An address field that identifies the location of the record in the Oracle database. This address field is called **Rowid.**

When data is inserted in the table, the Oracle engine 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 for the 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:

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

The ROWID format used by Oracle 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
Azurestaff.ora	3	30 MB
Azurestudent.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 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 zero. 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 **rowed** for the table record.

Retrieve order_no, order_date, client_no from sales_order table where client_no is equal to ‘C00001’. **There is no index on client_no created for the sales_order table.**

Table Name: Sales_order

Order_no	**Order_date**	**Client_no**
S00001	12-Nov-97	C00001
S00002	30-Nov-97	C00003
S00003	1-Dec-97	C00001
S00004	28-Dec-97	C00002
S00005	17-Jan-98	C00003
S00006	19-Jan-98	C00001

Example:

```
Select order_no,order_date,client_no From Sales_order Where
client_no=’C00001’;
```

When the above select statement is executed, since an index is not created on client_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. client_no= ‘C00001’ by comparing the value in the search criteria with the value in the client_no column from the first record to the last record in the table.

Table sales_order is indexed on client_no.

Since an index exists on the client_no column of the sales_order table, the index data will be represented as follows;

Index Name: idx_client_no

Client_no	**ROWID**
C00001	00000240.0000.00004
C00001	00000240.0002.00004
C00001	00000241.0002.00004
C00002	00000241.0000.00004
C00003	00000240.0001.00004
C00003	00000240.0001.00004

Note: The index is in the ascending order of client_no. The addresses have been assigned a data file number, block number and record number in the order of creation.

Example:

```
Select order_no,order_date,client_no From Sales_order Where
client_no='C00001';
```

When the above select statement is executed, since an index is created on client_no column, the Oracle engine will scan the index to search for a specific data value i.e. client_no equal to 'C00001'. The Oracle engine will then perform a sequential search to retrieve records that match the search criteria i.e. client_no= 'C00001'. When 'C00002' is read, the Oracle engine stops further retrieval from the index.

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

Client_no	**ROWID**
C00001	00000240.0000.00004
C00001	00000240.0002.00004
C00001	00000241.0002.00004

The Rowid in the current example indicates that the record with client_no 'C00001' is located in data file 0004. Two records are stored in block 00000240 with record number 0000 and 0002. The third record is stored in block 00000241 with record number 0002.

Thus, data retrieval from a table by using an index is faster then 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 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 it is called **Simple Index.** The syntax for creating simple index that allows duplicate values is:

Syntax:
```
Create Index <Index Name> On <Table Name>(ColumnName);
```

Example:
Create a simple index on client_no column of the Client_master table

```
SQL>Create Index idx_client_no On Client_master (Client_no) ;

Index Created.
```

Creating Composite Index:

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

Syntax:
```
Create Index <Index Name> On <Table Name>(ColumnName, ColumnName);
```

Example:
Create a composite index on the sales_order tables on column order_no and product_no.

```
SQL>Create Index idx_sales_order On Sales_order (Order_no,product_no) ;

Index Created.
```

Note: The indexes in the above examples do not enforce uniqueness i.e. the columns included in the index can have 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 **Simple Unique Index.** The syntax for creating a simple unique index is:

Syntax:
```
Create Unique Index <Index Name> On <Table Name> (Column Name);
```

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

Syntax:
```
Create Unique Index <Index Name> On <Table Name> (ColumnName,ColumnName);
```

Example:
Create a unique index on client_no column of the client_master table.

```
SQL>Create Unique Index idx_client_no On Client_master (Client_no);

Index Created.
```

Note: When the user defines a primary key or a unique key constraint, the Oracle engine automatically creates a unique index on the primary key or unique key column/s.

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

Syntax:
```
    Drop Index <Index Name>;
```

Example:
Remove index idx_client_no created for the table client_master.

```
    SQL>Drop Index idx_client_no;

    Index dropped.
```

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.

Object Relational Database

As of Oracle8, you can extend your relational database to include object-oriented concepts and structures. You will see an overview of the major object-oriented features available in Oracle8 and the impact they have on SQL.

Do I have to Use Objects?

Just because you use Oracle8 does not mean you have to use object-oriented programming (OOP) concepts when implementing your database. In fact, the database is referred to as an object-relational database management system (ORDBMS). The implication for developers is that three different "flavors" of Oracle are available:

Relational	:	The traditional Oracle relational database management system (RDBMS).
Object-relational	:	The traditional Oracle relational database, extended to include object-oriented concepts and structures such as abstract datatype, nested table, and varying arrays.
Object-oriented	:	An object-oriented database whose design is based solely on object-oriented analysis and design.

Oracle provides full support for all three implementations. If you have previously used Oracle as a relational database, you can use Oracle8 in the same manner.

Why Should I Use Objects?

Since you don't have to use objects, should you use them at all? At first, using OOP features may seem to complicate the design and implementation of your database systems – just as adding new features to any system may automatically increase its complexity.

Besides simplifying your interactions with data, objects may help you in other ways. Three benefits that come from using OOP features:

Object Reuse: If you write OOP code, you increase the chances of reusing previously written code modules. Similarly, if you create OOP database objects, you increase the chances that the database objects can be reused.

Standards Adherence: If you create standard objects, you increase the chance that they will be reused. If multiple applications or tables use the same set of database objects, you have created a effectively standard for the database objects. For example, if you create standard datatypes to use for all addresses, then all the addresses in your database will use the same internal format.

Defined Access Paths: For each object, you can define the procedures and functions that act upon it – you can unite the data and the method that access it. Having the access paths defined in this manner allows you to standardize the data access methods and enhance the reusability of the objects.

The costs of using objects are chiefly the added complexity of the system and the time it takes to learn how to implement the features. But, as you'll see in this chapter, the basics of extending the Oracle RDBMS to include OOP capabilities build easily upon the relational model presented in the earlier chapters. The short time required to develop and use abstract datatypes, should help you measure the time required for learning OOP features.

Oracle supports many different types of objects. In the following sections, you will see descriptions of the major object types available.

Abstract Datatypes

Abstract datatypes are datatypes that consist of one or more subtypes. Rather than being constrained to the standard Oracle datatypes of NUMBER, DATE, and VARCHAR2, abstract datatypes can more accurately describe your data. For example, an abstract datatype for addresses may consist of the following columns:

```
Street     Varchar2(50)
City       Varchar2(25)
StateChar(2)
Zip        Number
```

When you create a table that uses address information, you could create a column that uses the abstract datatype for addresses – and thus contains the Street, City, State, and Zip columns that are part of that datatype. Abstract datatypes can be nested; they can contain references to other abstract datatypes. You will see a detailed example of nested abstract datatypes.

Two of the benefits listed earlier for objects – reuse and standards adherence – are realized from using abstract datatypes. When you create an abstract datatype, you create a standard for the representation of abstract data elements (such as addresses, people, or companies). If you use the same abstract datatype in multiple places, then you can be sure that the same logical data is represented in the same manner in those places. The reuse of the abstract datatype leads to the enforcement of standard representation for the data.

You can use abstract datatypes to create object tables. In an object table, the columns of the table map to the columns of the abstract datatype.

Points to remember when using Object Types:

You cannot define constraints when defining object types. You can only use constraints on object but not on object types.

- You cannot insert data into object types.
- You can use this object type in an object and then perform regular RDBMS operations.
- You can use object types in object types or object tables.

Syntax:

```
Create Type <Type_Name> As Object
(ColumnName Datatype(Size),ColumnName Datatype (Size),……)
/
```

Example:

Create one abstract datatype and use it in object table.

```
SQL>Create Type Addr As Object(Street    Varchar2(50),City
Varchar2(25),State  Char(2),Zip Number)
/

Type created.

SQL>Create Table Employee(Empno number(3),Ename Varchar2 (20),Address
addr,Deptno Number(3),Salary Number(8,2));

Table created.
```

Insert records in to Employee table.

```
SQL>Insert into Employee Values(1127,’Tanmay’,addr(‘B/10 Sunil
Society’,’Ahmedabad’,380008),20,6500);

1 row created.
```

Nested Tables

A nested table is a table within a table. A nested table is a collection of rows, represented as a column within the main table. For each record within the main table, the nested table may contain multiple rows. In one sense, it’s a way of storing a one-to-many relationship within one table. Consider a table that contains information about Items, each of which may have many suppliers in progress at any one time. In a strictly relational model, you would create two separate tables – ITEM_DET and SUPPLIER_DET.

Nested tables allow you to store the information about suppliers within the ITEM_DET table. The

SUPPLIER_DET table records can be accessed directly via the ITEM_DET table, without the need to perform a join. The ability to select the data without traversing joins may make the data easier to access for users.

Even if you do not define methods for accessing the nested data, you have clearly associated the items and supplier data. In a strictly relational model, the association between ITEM_DET and SUPPLIER_DET tables would be accomplished via foreign key relationship.

Example:

```
SQL>Create or Replace Type Supp1 As Object
(Scode Number(3),Sname varchar2(20),Srate Number(8,2))
/
Type created.

SQL>Create or Replace Type Supp2 As Table of Supp1
/
Type created.
```

The **as table of** clause of this **create type** command tells Oracle that you will be using this type as the basis for a nested table. The name of the type, SUPP2, has a pluralized root to indicate that it stores multiple rows.

You can now create a table of ItemDet, using the SUPP2 datatype:

```
SQL>Create Table ItemDet
     (Itcode Number(3),Itname Varchar2(25),
 ItRate Number(8,2),Suppliers SUPP2)
Nested Table Suppliers store as Supp_Tab;

Table created.
```

When creating a table that includes a nested table, you must specify the name of the table that will be used to store the nested table's data. That is, the data for the nested table is not stored "inline" with the rest of the table's data. Instead, it is stored apart from the main table. Thus, the data in the Suppliers column will be stored in one table, and the data in another columns will be stored in a separate table. Oracle will maintain pointers between the tables. In this example, the "out-of-line" data for the nested table is stored in a table named SUPP_TAB.

Inserting Records into a Nested Table

You can **insert** records into a nested table by using the constructor methods for its datatype. For the Suppliers column, the datatype is SUPP2; thus, you will use the SUPP2 constructor method. The SUPP2 type, in turn, uses the SUPP1 datatype. As shown in the following example, inserting a record into ITEMDET table requires you to use both the SUPP2 and SUPP1 constructor methods. In the example, three suppliers are listed for the item named Bpl Tv.

Example:

```
SQL>Insert into ITEMDET values
     (1,'Bpl Tv',12000,
      SUPP2(
          SUPP1(101,'Abhay Shah',11500),
          SUPP1(102,'Himani',11300),
          SUPP1(103,'Brijesh',10800)
     ));
1 row created.
```

Querying Nested Tables

Nested tables support a great variety of queries. A nested table is a column within a table. To support queries of the columns and rows of a nested table, Oracle provides a special keyword, **THE.** To see how the **THE** keyword is used, first consider the nested table by itself. If it were a normal column of a relational table, you would be able to query it via a normal **select** command:

```
SQL>Select Sname /* This won't work.*/
from Supp2
 where scode=102;
```

SUPP2 is not a normal table; it's a datatype. To select columns (such as Sname) from the nested table, you first have to "flatten" the table so that it can be queried. That's where the **THE** function is used. First, select the nested table column from the main table:

```
SQL>Select * from THE(Select Suppliers from ItemDet
     Where Itcode=1);
```

Above command will display all the record of the nested table whose itcode is 1.

Using THE function insert record in nested table:

```
SQL>Insert into THE(Select Suppliers from ItemDet Where Itcode=1)
Values(104,'Jigar',11100);
1 row created.
```

Using THE function update records from nested table:

```
SQL>Update THE(Select Suppliers from ItemDet Where Itcode=1) Set
Sname='Tanmay' Where Scode=103;
1 row updated.
```

Using THE function delete records from nested table:

```
SQL>Delete From THE(Select Suppliers from ItemDet Where Itcode=1) Where
Scode=102;
1 row deleted.
```