Patni Computer Systems Ltd.

DBMS/SQL+

Training Workshop

Student Guide

Index

1. Introduction to Database

1.1. Introduction

A set of inter-related data is known as *database* and the software that manages it is known as *database management system* or DBMS. Hence DBMS can be described as "*a computer-based record keeping system which consists of software for processing a collection of interrelated data*". A set of structures and relationships that meet a specific need is called a *schema*.

A DBMS is a software that manages a collection of interrelated data elements, stored in a database, that can be accessed in a shared manner by a collection of applications programs. On a very conceptual level, we can think of a database as central reservoir of data that can be accessed by many users

A person known as the *Database Administrator* or the DBA centrally manages the database. The DBA initially studies the System and accordingly decides the types of data to be used, then the structures to be used to hold the data and the interrelationships between the data structures. He then defines data to the DBMS. The DBA also ensures the security of the database. The DBA usually controls access to the data through the user codes and passwords and by restricting the views or operations that the users can perform on the database.

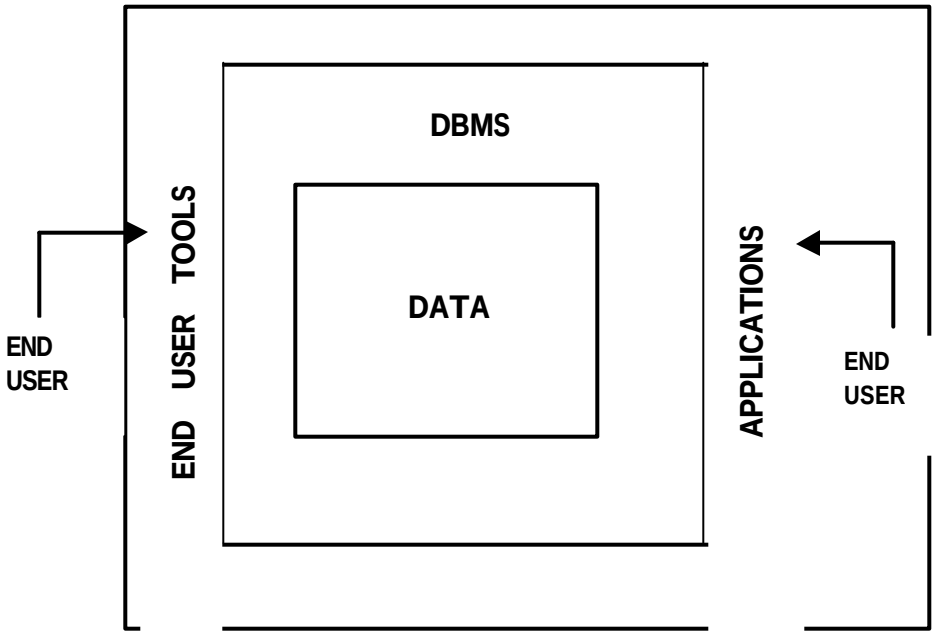


Figure 1-1 **DBMS Structure**

In today's environment, there must be one key individual who is responsible for the overall centralized control of the organization's data. We refer to the person as the *data base administrator* (DBA). It concerns policy-oriented tasks regarding data strategy and overall data planning. The DBA is responsible for the actual entities in the business environment that will be represented in the database. The DBA also works with end users and system analysts to document the data that will be stored in the database. The DBA initially studies the System and accordingly decides the types of data to be used, then the structures to be used to hold the data and the interrelationships between the data structures. He then defines data to the DBMS. The DBA also ensures the security of the database. The DBA usually controls access to the data through the user codes and passwords and by restricting the views or operations that the users can perform on the database.

1.2. Characteristics of DBMS

Some of the general characteristics of the DBMS have been discussed below:

1.2.1. Control of Data Redundancy

When the same data is stored in a number of files it brings in data redundancy. In such cases, if the data is changed at one place, the change has to be duplicated in each of the files.

The main disadvantages of data redundancy are**:**

- Storage space gets wasted
- Processing time may be wasted as more data is to be handled
- Inconsistencies may creep in

In the traditional file environment, application programmers design their own data files and establish their own record formats. This makes their programs dependent on the format of those records, and it tends to create a high degree of data redundancy. For example, many different programs in an installation may require access to information about employees. If the records that contain employee information are designed independently, the same piece of employee information may exist in different files, possibly in different forms. Through effective use of database techniques, this redundancy can be better controlled. Ideally, each piece of information is stored only once in the central pool, and all application programs that require the information gain access to it via the data base management system. By controlling redundancy in this manner, it is much easier for the installation to ensure that multiple copies of the same data element do not exist in different stages of updating.

1.2.2. Sharing of Data

DBMS allow many applications to share the data.

1.2.3. Maintenance of Integrity

Integrity of data refers to the correctness, consistency and interrelationship of data with respect to the application that uses the data. Some of the aspects of data integrity are:

- Many data items can only take a restricted set of values
- Certain field values are not to be duplicated across records. Such restrictions, called *primary key* constraints can be defined to the DBMS
- Data integrity, which defines the relationships between different files, is called *referential integrity* rule, which can also be specified to the DBMS

1.2.4. Support for Transaction Control and Recovery

Multiple changes to the database can be clubbed together as a single 'logical transaction'**.** The DBMS will ensure that the updates take place physically only when the logical transaction is complete**.**

1.2.5. Recovery and Restart

When traditional files are used to implement systems, recovery and restart routines (or programs) must be built into each individual application system. This makes it difficult to implement sophisticated recovery and restart procedures. With a database management system, recovery and restart procedures can be handled automatically on a system wide basis using the tools provided by DBMS software.

A DBMS provides powerful tools for automatically recovering from system failures, restoring the database to its original form, and restarting jobs that were affected by failures.

1.2.6. Data Independence

In conventional file based applications, programs need to know the data organisation and access technique to be able to access the data. This means that if you make any change in the way the data is organised you will also have to take care to make changes to the application programs that apply to the data. In DBMS, the application programs are transparent to the physical organization and access techniques.

1.2.7. Availability of Productivity Tools

Tools like querying language screen and report painter and other 4GL tools are available. These tools can be utilized by the end-users to query, print reports etc. SQL is one such language, which has emerged as standard**.**

1.2.8. Security

DBMSs provide tools by which the DBA can ensure security of the database.

1.2.9. Hardware Independence

Most DBMSs are available across hardware platforms and operating systems. Thus **the** application programs need not be changed or rewritten when the hardware platform **or** operating system is changed or upgraded.

1.3. The DBMS Models or Architectures

A database management system generally conforms to one of the three major database architectures: *hierarchical*, *network*, or *relational*. The range of data structures supported and the availability of data handling **languages** depend on the model of DBMS is based. The models are

1. The Hierarchical Model
2. The Network Model
3. The Relational Model.

1.3.1. The Hierarchical Model

In a hierarchically structured database, data records are connected with embedded pointers to form an inverted tree structure in which dependent records can have one and only one parent. A database model in which records in a file are associated in a one-to-many, or parent-child, relationship.

1.3.2. The Network Model

A network-structured database takes the form of a mesh structure, in which dependent records can have more than one parent. Like hierarchical model the relationship is implemented by means of embedded pointers. A DBMS model that builds a tight linkage, called a set, between elements of data.

1.3.3. The Relational Model

Student Table

Scode	Sname
S1	A
S2	B

Course Table

Ccode	Cname
C1	Physics
C2	Chemistry
C3	Maths
C4	Biology

Marks Table

Ccode	Scode	Marks
C1	S1	65
C2	S1	78
C3	S1	83
C4	S1	85
C3	S2	83
C4	S2	85

Figure 1-2 **Relational Tables**

With relation database, data is represented in the form of tables, and no embedded pointers are required to implement relationships between records. Because of lack of linkages relational model is easier to understand and implement.

1.4. Relational DBMS

1.4.1. Introduction

The relational model presents an orderly, predictable and intuitive approach to organizing, manipulating and viewing data.

1.4.2. RDBMS Terminology

Relational data consists of relations. A *relation* (or relational table) is a two dimensional table with special properties. A relational table consists of a set of named columns and an arbitrary number of rows. The columns are called ***Attributes or Fields*** and rows are called ***tuples or records***. Each attribute draws values from a pool of values called *domain*. Simply put, each attribute is associated with a domain. Note that it is possible to associate more than one attribute of a table(s) to a single domain.

DEPT table

Deptno	**Dname**	**Loc**
10	Accounting	New York
20	Research	Dallas
30	Sales	Chicago ← **'row' or 'tuple'**
40	Operations	Boston

↑

'column' or 'attribute'

EMPLOYEE table

Empno	**Empname**	**Job**	**Mgr**	**Deptno**
7369	Smith	Clerk	7902	20
7499	Allen	Salesman	7839	30
7566	Jones	Manager	7839	20
7839	King	President		10
7902	Ford	Analyst	7566	20

1.4.3. Properties of Relational Data Structures

Relational tables have six properties, which must be satisfied for any table to be classified as relational. These are:

1. *Entries of attributes are single valued or atomic*

   Entry in every row and column position in a table must be single valued. This means columns do not contain repeating groups. For example: There is an attribute in an **EMPLOYEE** table called **Empname,** which should not contain values like Smith John as it is not atomic or single value. Instead you can decompose the attribute **Empname** to **Firstname**, **Middlename** and the **Lastname.** Now each of these attributes will contain single value

2. *Entries of attribute are of the same kind or data type.*

   Entries in a column must be of same kind or data type. A column supposed to store salary of an employee should not store commission. For instance, all the values of the attribute **MGR** in the table **EMPLOYEE** should contain only digits not any other characters.

3. *No two rows are identical*

   Each row should be unique, this uniqueness is ensured by the values in a specific set of columns called the ***primary key*****.**

4. *The order of attributes is unimportant*

   There is no significance attached to order in which columns are stored in the table. A user can retrieve columns in any order.

5. *The order of rows is unimportant*

   There is no significance attached to the order in which rows are stored in the table. A user can retrieve rows in any order.

6. *Every column can be uniquely identified.*

   Each column is identified by its name and not its position. A column name should be unique in the table.

1.4.4. Database Schema

Database schema is the organization of information within a database for a single user or multiple users. It is a store of data that describes the content and structure of the physical data store. It contains various information like data types, relationships, access controls, etc. The Schema for a user is a set of structures and relationships between them that meet a specific need. It is a repository that has information about the structure and content of the database. It also contains information about how the data is stored internally, as well as how it is stored physically on the storage device.

1.4.5. Data Integrity

Integrity refers to the wholeness and soundness of the database. Some of the most important integrities are discussed below.

1.4.6. Domain Constraints

A ***domain*** is the set of all possible data values of some particular type. For example, the domain of **employee** numbers is the set of all valid employee numbers; the domain of employee names is the set of valid letters. Thus, domains are *pools of values*, from which the actual values appearing in attributes (columns) are drawn.

1.4.7. Primary Key and Entity Integrity

It is also possible, though again unusual, for a table to have *more than one* unique identifier. In such a case we would say that the table has multiple *candidate* keys; we would then choose one of those candidate keys to be the *primary* key, and the remainder would then be said to be *alternate* keys (an alternate key is thus a candidate key that is not primary key).

Let CK be some subset of the columns of table T. Then CK is a *candidate key* for T if and only if satisfies the following property:

Uniqueness: At any give time, no two rows of T have the same value for CK

Note that the relational model *requires* every table to have at least one candidate key, and hence *requires* every table to have a primary key. This requirement is equivalent to the requirement that, at any given time, no two rows in the table are identical.

Primary key is a column or set of columns in a table, which uniquely identifies a row in a table. No two rows of the table can have the same values for the primary key. Entity integrity is maintained by ensuring that none of the columns that make up the primary key can take "NULL" (unknown) values.

1.4.8. Foreign Key and Referential Integrity

The referential rule simply states that the database must not contain any unmatched foreign key values (i.e., nonnull foreign key values for which there does not exist a matching value of the corresponding primary key).

In the **EMPLOYEE** table the **deptno** field can be considered as a **'foreign key'**. This means that the **EMPLOYEE** table cannot contain a value for the **deptno**, which does not exist in the table **DEPT**.

The referential integrity rule specifies that if a foreign key in table **A** refers to the primary key in table **B**, then every value of the foreign key in table **A** must be null or must be available in table **B**.

1.4.9. Delete Restrict Referential Integrity

The delete is “restricted” to the case where there are no matching rows in the table T2 (it is rejected if any such rows exist).

This means that no primary key value can be deleted if there are some foreign key values dependent on it. i.e Dept no 10 cannot be deleted because there are some employees assigned to deptno 10.

1.4.10. Delete Cascade Referential Integrity

This means that if a primary key value is deleted then all foreign key values dependent on it will be deleted. i.e. if Dept no 10 is deleted then all rows (employees) having dept no 10 will be deleted.

1.4.11. Update Cascade referential Integrity

This means that if a primary key value is updated then all foreign key values dependent on it will be updated to the new value of primary key. i.e. if we change the deptno 10 to 50 then all the employees in deptno 10 will be also shifted to deptno 50. (Column deptno will be automatically updated)

1.4.12. Column Constraints

These are the constraints, which specify restrictions on the values a column can take. These restrictions may be defined with or without other values in the same row

2. Introduction to SQL

2.1. What is SQL ?

SQL or **Structured Query Language** is the set of commands, which is used for interacting with Relational Databases. ANSI has standardized the commands and all relational databases conform to this standard. Most RDBMS provide extensions to it to make the life of the application programmer easier.

2.1.1. Components of SQL

SQL can be broadly classified into three components.

1. **Data Definition Language (DDL):**

   This component is used for defining and destroying objects in your application like Tables, Indexes etc. E.g. DDL statements like CREATE TABLE

2. **Data Manipulation Language (DML):**

   This is used to modify and retrieve the data according to your needs for example Inserting, Updating and Querying. E.g. DML statements like SELECT, INSERT, DELETE and UPDATE.

3. **Data Control Language (DCL):**

   This component is used for defining your security for your application objects like Tables etc. E.g. DCL statements like GRANT and REVOKE.

A *database* is a collection of structures with appropriate authorizations and accesses defined. The tables, indexes are structures in the database are called ***objects*** in the database; their names and those of columns are called ***identifiers*****.**

SQL commands are the instructions given for some operation on the database and the fact that SQL DML like SELECT, UPDATE and DELETE statements typically operate on *entire sets of rows*, instead of just one row at a time. Hence, SQL is called *set-level language*.

2.1.2. Benefits of SQL

The benefits of SQL are as follows:

1. Non-Procedural Language:
   - Set-level languages such as SQL are sometimes described as "nonprocedural," on the grounds that the users specify what, not how (i.e., they say what data they want without specifying a procedure for getting it). Perhaps a better way of putting matters is to say that languages like SQL are at a *higher level of abstraction* than languages like COBOL or C.
   - It processes sets of records rather than just one record at a time.
   - It provides automatic navigation to the data. The process of "navigating" around the physical database to locate the desired data is performed automatically by the system, not manually by the user
2. Language For All Users:
   - System Administrators
   - Database Administrators
   - Security Administrators
   - Application Programmers
   - End-Users

2.1.3. Data types Supported in SQL

Oracle 8 has evolved from an RDBMS to an ORDBMS (Object Relational Database Management System). The traditional Data Types are called *Scalar Data Types*.

SCALAR DATA TYPES

Datatype	Description
CHAR(n)	To store fixed length string Maximum length = 2000 bytes Eg. name CHAR(15)
VARCHAR2(n)	To store variable length string Maximum length = 4000 bytes Eg. description VARCHAR2(100)
LONG(n)	To store variable length string Maximum length = 2 Gigabytes Eg. synopsis LONG(5000)

NUMBER(p,s)	To store numeric data Range is 1E-129 to 9.99E125 Max Number of significant digits = 38 Eg. salary NUMBER(9,2)
DATE	To store DATE. Range from January 1, 4712 BC to December 31, 9999 AD. Both DATE and TIME are stored. Requires 7 bytes. Eg. hiredate DATE
RAW(n)	To store data in binary format such as signature, photograph. Maximum size = 255 bytes
LONG RAW(n)	Same as RAW. Maximum size = 2 Gigabytes

2.2. Creation of Database Objects

2.2.1. TABLE

Tables are objects, which store the user data. In Oracle 8i, tables can be permanent or temporary. First we will see permanent tables. In the next section, temporary tables are explained.

Syntax

```
CREATE TABLE table_name
 (
 {col_name.col_datatype [[CONSTRAINT
 const_name][col_constraint]]},...
 [table_constraint],...
 )
 [AS query]
```

A table can have a maximum of 1000 columns. Only one column of type LONG is allowed per table.

Table_name,col_name,const_name	A string upto 30 characters length. Can be made up to A-Z,0-9,$,_,# Must begin with a non-numeric ORACLE data characters.
Col_datatype	One of the previously mentioned types
Col_constraint	A restriction on the column Can be of following types- PRIMARY KEY, NOT NULL, UNIQUE, FOREIGN KEY, CHECK, Can be named

	Only one PRIMARY KEY allowed per table.
Table_constraint	A restriction on single or multiple columns. The types are same as in col_constraint. (NULL constraint is not allowed here).
AS query	Query is an SQL statement using SELECT command. SELECT command returns the rows from tables. See SECTION 3.11 for more information on SELECT command. Useful if the table being created is based on an existing table. New table need not have all the columns of the old table. Name of columns can be different. All or part of the data can be copied.

Any object created by a user is accessible to the user and the DBA only. To make the object accessible to other users, the creator or the DBA must explicitly give permission to others. SEE Section 3.12 for granting permissions.

Example:

```
CREATE TABLE emp
(
empno          NUMBER(4),
ename          VARCHAR2(10) ,
deptno         NUMBER(2) ,
job            CHAR(9) ,
hiredate       DATE  );
```

If a table is created as shown above, then there is no restriction on the data that can be stored in the table. However, if we wish to put some restriction on the data, which can be stored in the table, then we must supply some constraints for the columns.

For example, if we want

1. To make empno as the primary key of the table and
2. To ensure that the ename column does not contain NULL values and
3. The job column to have only UPPERCASE entries and
4. to put the current date as the default date in hiredate column in case data is not supplied for the column.

The create statement can be rewritten as:

```
CREATE TABLE emp
(
   empno     NUMBER(4)    CONSTRAINT P_KEY PRIMARY KEY,
   ename     VARCHAR2(10) CONSTRAINT ENAME_NOT_NULL  NOT NULL,
   deptno    NUMBER(2),
   job       CHAR(9)      CONSTRAINT JOB_ALL_UPPER
                          CHECK(job = UPPER(job)),
   hiredate  DATE         DEFAULT SYSDATE
);
```

Now if we have a table DEPT that has following description,

```
CREATE TABLE Dept
 (
    Deptno    NUMBER(2) ,
    Dname     VARCHAR2(14)  NOT NULL ,
    Loc       VARCHAR2(13)  NOT NULL
 );
```

In EMP table, for deptno column, if we want to allow only those values that already exist in deptno column of DEPT table, we must enforce what is known as REFERENTIAL INTEGRITY.

To enforce *referential integrity*, declare deptno field of DEPT table as PRIMARY KEY and deptno field of EMP table as FOREIGN KEY as follows.

```
CREATE TABLE Dept
 (
    Deptno  NUMBER(2)    CONSTRAINT  DEPTNO_P_KEY PRIMARY KEY,
    Dname   VARCHAR2(14)  NOT NULL,
    Loc     VARCHAR2(13)  NOT NULL
  );
```

```
CREATE TABLE Emp
  (
    Empno    NUMBER(4)     CONSTRAINT P_KEY PRIMARY KEY,
    Ename    VARCHAR2(10)  CONSTRAINT ENAME_NOT_NULL  NOT NULL,
    Deptno   NUMBER(2),
    Job      CHAR(9)       CONSTRAINT JOB_ALL_UPPER
                          CHECK (Job =UPPER(Job)),
    Hiredate DATE DEFAULT SYSDATE,
    CONSTRAINT DEPTNO_F_KEY FOREIGN KEY (Deptno)
               REFERENCES Dept(Deptno)
  );
```

In above example FOREIGN KEY has been declared as a table constraint. Same can be given as a column constraint as follows.

```
CREATE TABLE emp
(
  Empno     NUMBER(4)     CONSTRAINT P_KEY PRIMARY KEY,
  Ename     VARCHAR2(10) CONSTRAINT ENAME_NOT_NULL  NOT NULL,
  Deptno    NUMBER(2)     CONSTRAINT DEPTNO_F_KEY REFERENCES
                          Dept(Deptno),
  Job       CHAR(9)       CONSTRAINT JOB_ALL_UPPER
                          CHECK(job=UPPER(job)),
  Hiredate DATE DEFAULT SYSDATE
);
```

The following example shows how to create a new table based on an existing table

```
CREATE TABLE Newemp(Emp#,Emp_Name,Hire_Date)
AS
(
   SELECT Empno,Ename,Hiredate FROM emp
   WHERE Hiredate >'01-jan-82'
);
```

A new table will be created. It will contain empno, ename, hiredate of all employees hired after 1st Jan 1982. Constraints on an old table will not be applicable for a new table

2.3. Modification of Existing Database Objects

2.3.1. TABLE

An existing table can be modified even if contains data. However, it is recommended that you create a table after putting sufficient thought into it.

```
ALTER TABLE table_name
   [ADD (col_name col_datatype col_constraint ,...)]|
   [ADD (table_constraint)]|
   [DROP CONSTRAINT constraint_name]|
   [MODIFY existing_col_name new_col_datatype  new_constraint new_default]
   [DROP COLUMN existing_col_name]
   [SET UNUSED COLUMN existing_col)name];
```

table_name must be an existing table.

The rules for adding a column to the table are:

1. Can add any column without a NOT NULL specification.
2. Adding a NOT NULL column is possible in three steps.

- Add a column without NULL specification
- Filling every row in that column with data
- Modifying the column to be NOT NULL

The rules for modifying the columns are:

1. Can increase a character column's width any time.
2. Can increase the number of digits in a number at any time.
3. Can increase or decrease the number of decimal places in a number column at any time. Any reduction on precision and scale can be on empty columns only.
4. You can add only NOT NULL constraint using column constraints. Rest all constraints have to be specified as table constraints

For adding three more columns to the EMP table.

```
ALTER TABLE emp
   ADD (sal NUMBER(7,2)  CONSTRAINT  SAL_GRT_0 CHECK(sal >0),
        mgr NUMBER(4),
        comm NUMBER(9,2));
```

For adding referential Integrity on mgr column

```
ALTER TABLE emp
  ADD CONSTRAINT EMP_FMGR_KEY FOREIGN KEY (mgr) REFERENCES
  EMP (empno);
```

For modifying the width of sal column.

```
ALTER TABLE emp
    MODIFY  (sal NUMBER (8,2))  ;
```

For dropping the FOREIGN KEY constraint on mgr

```
ALTER TABLE emp
   DROP CONSTRAINT EMP_F_KEY;
```

Rules for dropping column

Oracle 8i allows you to drop a column from a table. This can be done in two ways.

1. Marking the columns as unused and then later dropping them

```
ALTER TABLE emp
   SET UNUSED COLUMN comm;

ALTER TABLE emp
   SET UNUSED (sal, hiredate);
```

Once all the required columns have been marked as unused, we can use the following command to remove the columns permanently. Columns once marked as unused cannot be recovered. Marking the columns as unused does not release the space occupied by them back to the database. Also until you drop these columns actually, they continue to count towards the absolute limit of 1000 columns per table. Also, if you mark a column of data type LONG as UNUSED, you cannot add another LONG column to the table until you actually drop the unused LONG column.

```
ALTER TABLE emp
  DROP UNUSED COLUMNS;
```

The advantage of the above-mentioned steps is that marking the columns is much faster process than dropping the columns.

You can refer to the data dictionary table USER_UNUSED_COL_TABS to get information regarding the tables with columns marked as unused.

2. Dropping the columns directly

```
ALTER TABLE emp
   DROP COLUMN sal;
```

This command should be used with caution.

Before the user issues ***DROP COLUMN*** command, the user must have exclusive control over the table.

When you drop a column

- All indexes defined on any of the target columns are also dropped
- All constraints that reference a target column are removed

2.4. Deletion Of Database Objects

Deletion of objects existing in the database is an easy task. Just say

```
DROP Obj_Type obj_name;
```

For example

```
DROP TABLE EMP;
```

A table that is dropped cannot be recovered. When a table is dropped, dependent objects such as indexes are automatically dropped. *Synonyms* and *views* created on the table remain, but give an error if they are referenced. You cannot delete a table that is being referenced by another table. To do so use the following

```
DROP TABLE table-name CASCADE CONSTRAINTS;
DROP TABLE dept CASCADE CONSTRAINTS;

DROP SYNONYM new_emp;
```

If new_emp is a synonym for a table, then the table is not affected in any way. Only the duplicate name is removed.

2.5. Addition Of Data into Tables

```
INSERT INTO table_name[(col_name1,col_name2,...)]
    {VALUES (value1,value2,....) | query};
```

- If values are specified for all columns in the order specified at creation, then col_names can be omitted
- Values should match datatype of the respective columns
- Number of values should match the number of column names mentioned
- All columns declared as NOT NULL should be supplied with a value
- Character strings should be enclosed in quotes
- DATE values should be enclosed in quotes
- VALUES will insert one row at a time
- Query will insert all the rows returned by the query
- Table_name can be a table or a view or a synonym name
- If table_name is a view, then following restrictions apply
- View CANNOT have a GROUP BY, CONNECT BY, START WITH, DISTINCT, UNION, INTERSECT OR MINUS clause or a join
- If view has WITH CHECK OPTION clause, then a row that cannot be returned by the view will not be inserted

Inserting a row in emp table giving all values

```
INSERT INTO emp
 VALUES (s1. NEXTVAL,'JONATHAN SEAGULL',25,
            '615-3520',007,'01-APR-1993',7500);
```

- 007 is a dept number which exists in dept table
- s1.NEXTVAL returns the next serial number from sequence s1

Inserting a row in EMP table giving some values.

```
INSERT INTO EMP  (empno, ename, sal)
 VALUES (200, 'Archer', 40000);
```

- This row will be created if all the constraints like NOT NULL are satisfied.

Inserting rows in a table from another table.

```
INSERT INTO new_emp_table
       SELECT * FROM emp WHERE emp.hiredate > '11-jan-82';
```

This example assumes that new_emp_table exists.

2.5.1. Using Substitution Variables

Instead of using a table name or a column name or a value in the SQL, we can use a variable.

INSERT INTO dept VALUES (&deptno, '&dname', '&loc');

When the above INSERT statement is executed from the SQL prompt, the user is prompted with following

Enter value for deptno: 10
Enter value for dname: Sales
Enter value for loc: Mumbai

After entering the above details SQL displays the following

old 1: INSERT INTO dept VALUES(&1,&2,&3)
new 1: INSERT INTO dept VALUES(10,'Sales','Mumbai')
INSERT INTO dept VALUES (10,'Sales','Mumbai')

Deptno, dname and loc variables values will be obtained at runtime and substituted in the SQL statement before execution of the statement. The

ampersand & before the variable name indicates that, what follows is a variable and not a value.
Since there will be direct substitution, quotes are not required when you give input to dname and loc as quotes are already present. If the statement was given as

INSERT INTO dept VALUES (&deptno, &dname, &loc);
then while giving input quotes are required.

2.6. Modification of Existing Data in Tables

```
UPDATE table_name
   SET  col_name = value|
        col_name = SELECT_statement_returning_single_value|
        (col_name,...) = SELECT_statement
  [WHERE condition];
```

Provides automatic navigation to the data. If WHERE is omitted, all the rows will be updated and if WHERE is specified only those rows that satisfy the condition will be updated.

```
UPDATE emp
   SET ename = 'ADAM'
   WHERE ename = 'ADAMS';
```

For making salary of SMITH equal to that of employee no 7369.

```
UPDATE emp
   SET SAL = (SELECT SAL FROM emp
                 WHERE empno = 7369)
   WHERE ename = 'SMITH';
```

All employees other than the PRESIDENT will have the same job, salary and commission as that of ALLEN.

```
UPDATE emp
       SET (job,sal,comm) = (SELECT job,sal,comm FROM emp
                                WHERE ename = 'ALLEN')
       WHERE NOT job = 'PRESIDENT';
```

2.7. Deletion of Data from Tables

```
DELETE [FROM] {table_name | alias }
     [WHERE condition];
```

table_name can be a table or a view.

If WHERE is omitted, all rows from the table are removed and if WHERE is specified all rows, which satisfy the condition, are removed.

FROM can be omitted without affecting the statement.

```
DELETE FROM emp;     -- all rows will be deleted

DELETE FROM emp
       WHERE deptno = 007;
```

Rows of employees attached to dept 007 will be deleted.

```
DELETE emp
       WHERE ename = 'JOHN';
```

Record of 'JOHN' will be deleted.

2.8. Transaction Control

A *transaction* is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit; the effects of all the SQL statements in a transaction can be either all *committed* (applied to the database) or all *rolled back* (undone from the database).

Banking is the standard example. A transaction would encompass the transfer of money from one account to another. The transfer is a transaction because the debit of money from one account and credit to another must be performed as a unit--one part cannot fail.
A transaction is not necessarily just one SQL operation; rather, it is a *sequence* of several operations, in general, that transforms a consistent state of the database into another consistent state, without necessarily preserving consistency at all intermediate points.

A transaction begins with the first executable SQL statement. A transaction ends when any of the following occurs:

- You issue a COMMIT or ROLLBACK statement
- You execute a DDL statement (such as CREATE, DROP, RENAME, ALTER). If the current transaction contains any DML statements, Oracle first commits the transaction, and then executes and commits the DDL statement as a new, single statement transaction
- A user disconnects from the session. (The current transaction is committed.)
- A user process terminates abnormally. (The current transaction is rolled back.)

2.8.1. COMMIT

Committing a transaction makes permanent the changes resulting from all successful SQL statements in a transaction. When a transaction is committed, the following events take place:

- The changes made by the current transaction are made permanent
- The locks acquired by the transaction are released
- The transaction is marked as "complete"

Syntax: COMMIT;

2.8.2. The SAVEPOINT Command

- Savepoints name and mark the current point in the processing of a transaction
- Savepoints allow to undo parts of a transaction instead of the whole transaction
- If an error is made in between, it is not necessary to restart the transaction from the beginning
- The number of active Savepoints per user session is 255

Example:

```
INSERT into emp(empno,ename,sal)
        VALUES(1004,'Amit',5000);

SAVEPOINT spt1;

UPDATE emp
        SET sal=sal+1000
        WHERE deptno=10;

ROLLBACK to spt1;
```

The SAVEPOINT command divides a transaction into smaller sub units or logical units. Very large, complex transactions might require dozens or hundreds of DML commands, it may help your application design if you can hold at certain points and ROLLBACK or COMMIT just one part of the total transaction, an example of this is a complex user information input such as entering mathematical modeling data. The SAVEPOINT command has a parameter, which is the name of the savepoint. This name can be used to rollback all updates from the time

when you issued the named savepoint. You can reuse the savepoint name, each time you redeclare an existing savepoint the old savepoint of that name is deleted.

2.8.3. The ROLLBACK Command

Changes made to the database without COMMIT may be abandoned using the ROLLBACK statement. When a transaction is rolledback, it is as if the transaction never occurred. When a transaction is rolled back, the following events take place

- Any changes made to the database is undone
- The locks acquired by the transaction are released
- The transaction is ended

Syntax: ROLLBACK [TO SAVEPOINT <savepoint name>];

DELETE FROM Dept WHERE deptno = 10

SAVEPOINT delete_finished

INSERT INTO Dept VALUES(50,'Mkt','Michigan');

ROLLBACK TO delete_finished

ROLLBACK

The ROLLBACK command removes pending database updates. Issuing a ROLLBACK without specifying a savepoint name removes all pending updates since the last transaction start event whereas issuing a ROLLBACK with a savepoint name parameter removes all pending updates since the time the specified savepoint was created. The first ROLLBACK above removes all updates associated with the INSERT command. The second ROLLBACK removes all updates associated with the DELETE command.

2.9. How to Retrieve Data from Tables ?

SELECT command is used to retrieve rows from table or views. Rows can be retrieved from a single table or multiple tables.

The syntax of the SELECT statement is

```
SELECT [ALL | DISTINCT] { * | col_name,...}
```

```
FROM table_name alias,...
  [ WHERE expr1 ]
  [ CONNECT BY expr2 [ START WITH expr3 ] ]
  [ GROUP BY expr4 ] [ HAVING expr5 ]
  [ UNION | INTERSECT | MINUS SELECT ... ]
  [ ORDER BY expr | ASC | DESC ];
```

Each clause is evaluated on the result set of a previous clause. The final result of the query will be always a result table. Only FROM clause is essential. The WHERE, GROUP BY, HAVING, ORDER BY, UNION are optional.
All the examples that follow are based on EMP and DEPT tables that are already available to you. For more information about these tables, refer **Appendix B**.

To select all rows from table dept.

```
SELECT deptno,dname,loc FROM dept;
```

```
DEPT NO    DNAME              LOC
-------    ------------        --------
10         ACCOUNTING         NEW YORK
20         RESEARCH           DALLAS
30         SALES              CHICAGO
40         OPERATIONS         BOSTON
```

- It is NOT necessary to retrieve all the COLUMNS of the table. If LOC is not required, then the following query can be used

```
SELECT deptno,dname FROM dept;
```

- It is NOT necessary to list the column names in the same order as done at the time of table creation. If dept name is to be displayed followed by dept number, then use

```
SELECT dname,deptno FROM dept;
```

- If all columns are to be retrieved, then instead of listing the column names, use. The star or asterisk is shorthand for a list of all column names in the table(s) named in the FROM clause, in the left-to-right order in which those columns appear in the relevant tables(s)

```
SELECT * FROM dept;
```

- When rows are retrieved and displayed, the column heading is same as the column name. To change the heading from LOC to LOCATION, use

```
SELECT dname,loc "LOCATION" FROM dept;
```

- To list deptno from EMP table

```
SELECT deptno FROM emp ;

DEPTNO
------------
20
30
30
20
30
30
10
20
10
…
14 rows selected
```

DISTINCT Parameter

In the above output, some of the values have been repeated. By default, all values are retrieved. If you wish to remove duplicate values, then say

```
SELECT DISTINCT deptno FROM emp;
```

Note: It just so happens in this particular example that each row contains a single value; the effect of the DISTINCT specification is therefore to eliminate duplicate values. In general, however, DISTINCT means "eliminate duplicate rows.

2.9.1. WHERE Clause

WHERE clause is used to do selective retrieval of rows. It follows FROM clause and specifies the search condition. The result of the WHERE clause is the row or rows retrieved from the tables which meet the search condition.

The syntax is of the form

```
WHERE  <search condition>
```

COMPARISON Predicate

The comparison predicates specifies comparison of two values. It is of the form

```
< Expression> < operator >  < Expression>
< Expression> <operator> <subquery>  --- Will be discussed Later
```

The operators used are

=	Equal to
<>	not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

To list all employees working in department number 10 from EMP table

```
    SELECT ename "NAME", deptno  FROM emp
           WHERE deptno = 10;
NAME                DEPTNO
----------- ------------
CLARK               10
KING                10
MILLER              10
```

To list all employees whose sal is greater than or equal to 2000

```
    SELECT ename,sal from emp where sal >=2000;
ENAME               SAL
----------  ----------
CLARK               2450
BLAKE               2850
JONES               2975
SCOTT               3000
FORD                3000
KING                5000
```

BETWEEN Predicate – Retrieval Using BETWEEN

```
    <expression> BETWEEN  < expression> AND <expression>
```

A BETWEEN predicate X BETWEEN Y AND Z is really same as
X >= Y AND X <= Z

The BETWEEN condition is really just shorthand for a condition involving two individual comparisons "ANDed" together.

- To list names and sal of all employees whose sal is between 2000 and 3000 including 2000 and 3000

```
SELECT ename,sal FROM emp WHERE sal  BETWEEN 2000 AND 3000;
```

```
ENAME             SAL
----------        ----------
CLARK             2450
BLAKE             2850
JONES             2975
SCOTT             3000
FORD              3000
```

To list all employees hired during the year '82.

- ```
  SELECT ename, hiredate, sal FROM emp
       WHERE hiredate BETWEEN '01-jan-82' AND '31-dec-82';
  ```

```
ENAME             HIREDATE
-----------       ----------------
SCOTT             09-DEC-82
MILLER            23-JAN-82
```

IN Predicate – Retrieval Using IN

```
<Expression> IN <LIST>
<Expression> IN <SUBQUERY>
```

IN, like BETWEEN, is really just shorthand. An IN condition is logically equivalent to a condition involving a sequence of individual comparisons all “ORed” together.

The data types should match. On right hand side a subquery can be used.

- To get all the employees from deptno 10 and 20

```
SELECT empno, ename, sal, deptno FROM emp
       WHERE deptno in (10, 20);
```

```
EMPNO       ENAME       SAL         DEPTNO
----------  ---------- ---------- ------------
7369        SMITH       800         20
7566        JONES       2975         20
7782        CLARK       2450        10
7788        SCOTT       3000        20
7839        KING        5000        10
7876        ADAMS       1100         20
7902        FORD        3000        20
```
```

```
7934        MILLER      1300        10
```

LIKE Predicate –Retrieval Using LIKE

<COLUMN > LIKE < PATTERN>

The pattern contains a search string along with other special characters % and _. The % character (percent) represents *any sequence of n characters* (where n may be zero) where as the character _ (underscore) represents *any single character*. A pattern %XYZ% means search is to be made for string XYZ in any position. A pattern '_XYZ%' means search is to be made for a string XYZ in position 2 through 4 with any single character in position 1 and 0 or more characters starting from position 5.

To search for characters % and _ in the string itself we have to use an escape character. For example to search for string NOT_APP in column status we have to use the form
Status like 'NOT_APP' ESCAPE \' The use of \ as escape character is purely arbitrary.

- To list all employees whose name begins with 'J'

```
SELECT ename  FROM emp WHERE ename LIKE 'J%';
ENAME
------------
JONES
JAMES
```

- To list employees whose name begins with 'J' and has 'N' as the 3rd CHARacter, say

```
SELECT ename  FROM emp    WHERE ename LIKE 'J_N%';
ENAME
----------
JONES
```

NULL Predicate –Retrieval Involving NULL

The NULL SQL keyword is used to represent either a missing value or a value that is not applicable in a relational table. A null can be assigned but it cannot be equated with anything.

The NULL predicate specifies a test for null values. The form **for NULL predicate is:**

```
< COLUMN SPECIFICATION > IS NULL.
< COLUMN SPECIFICATION > IS NOT NULL.
< COLUMN SPECIFICATION > IS NULL
```

returns true only when column has null values.

```
<COLUMN> = NULL CANNOT BE USED TO COMPARE NULL VALUES.
```

- To list all employees not entitled for commission.

```
SELECT ename, sal, comm FROM emp WHERE comm IS NULL;
```

```
ENAME       SAL         COMM
----------  ---------   -----------
SMITH        800
JONES       2975
BLAKE       2850
CLARK       2450
SCOTT       3000
KING        5000
ADAMS       1100
JAMES        950
FORD        3000
MILLER      1300
```

- To list all employees who receive commission

```
SELECT ename,sal,comm  FROM emp WHERE comm  IS NOT NULL;
```

```
ENAME       SAL          COMM
----------              ----------      ----------
ALLEN        1600  300
WARD         1250    500
MARTIN       1250        1400
TURNER       1500     0
```

Even though TURNER has not received any comm, the row has been retrieved because it contains a 0 and not a NULL. NULL is a special representation without any value.

- To list the BIG BOSS of the company, except for BIG BOSS, all employees have a value in the MGR column.

```
SELECT ename, job,salary  FROM emp WHERE mgr IS NULL;

ENAME         JOB               SAL

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

KING          PRESIDENT         5000
```

Few More Predicates

Predicates EXISTS, ANY, ALL will be discussed in SUBQUERY sections.

Combining Predicates Using Logical Operators

Above predicates can be combined using logical operators AND, OR, NOT. The evaluation proceeds from left to right and order of evaluation is enclosed in parenthesis

- To list employees working in dept NUMBER 10 or 20, say

```
SELECT ename "NAME",deptno
   FROM emp
   WHERE deptno = 10 OR deptno = 20;
```

- To list employees hired after 01/09/81 and working in dept NUMBER 10, use

```
SELECT ename ,hiredate,deptno FROM emp
    WHERE deptno = 10 AND hiredate >'01-SEP-81';
```

- To list employees working in depts other than 10, use

```
SELECT ename FROM emp
       WHERE NOT deptno = 10;
```

NOT is a negation operator.

Aggregate Functions

Although quite powerful in many ways, the SELECT statement as so far described is still inadequate for many practical problems. For example, even a query as simple as "How many employees are there?' cannot be expressed using the constructs introduced up till now. SQL therefore provides a number of special *aggregate* (or *column*) functions to enhance its basic retrieval power. Some of the aggregate functions available are COUNT, SUM, AVG, MAX and MIN. Apart from the special case of "COUNT(*)", each of these functions operates on the collection of scalar values in one column of some table.

SQL provides a set of built-in functions for producing a single value for an entire group. These functions are called ***Set functions*** or ***Aggregate functions***. These functions can work on a normal result table or a grouped result table. If the result is not grouped then the aggregate will be taken for the whole result table.

1. COUNT(*)

COUNT returns the number of rows in a table.

To find the total number of employees

```
SELECT COUNT(*)  FROM emp;
COUNT(*)
---------------
14
```

It is possible to restrict the rows over which COUNT operates.

To find the total number of CLERKS, use

```
SELECT COUNT(*) FROM emp WHERE job = 'CLERK' ;
```

To find the total number of CLERKS hired after '13-jan-81', say

```
SELECT COUNT(*) FROM emp
        WHERE job = 'CLERK' AND hiredate >'13-jan-81' ;
```

When an aggregate function is used in a SELECT statement, column names cannot be used in SELECT unless GROUP BY clause is used.

2. SUM(COL_NAME | EXPRESSION)

SUM returns the total of values present in a particular column or a number of columns linked together in the expression. All the columns, which form the argument to SUM, must be numeric only.

To find the sum paid as salary to all employees every month

```
SELECT SUM(sal) FROM emp;

SUM(SAL)
---------------
29025
```

To find the yearly compensation paid to all SALESMEN use

```
SELECT SUM(12*sal)  FROM emp
    WHERE job = 'SALESMAN';
```

3. AVG (COL_NAME | EXPRESSION)

AVG is similar to SUM. AVG returns the average of the values in the column. The restrictions, which apply on SUM also, apply on AVG.
To find the average salary of all employees

```
SELECT AVG(sal)    FROM emp;
```

```
AVG (SAL)
---------------
2073.21429
```

To find the average yearly compensation paid to SALESMEN, use

```
SELECT AVG(12*sal)  FROM emp
      WHERE job = 'SALESMAN';
```

4. MIN (COL_NAME | EXPRESSION)

MIN returns the smallest value in the column. MIN accepts columns, which are NON-NUMERIC too.
To find the minimum salary paid to any employee

```
SELECT MIN(sal) FROM emp;
```

```
MIN(SAL)
--------------
800
```

To list the employee who heads the list alphabetically, use

```
SELECT MIN(ename) FROM emp;
```

5. MAX (COL_NAME | EXPRESSION)

MAX is the reverse of MIN. MAX returns the maximum value from among the list of values in the column.

To find the maximum salary paid to any employee.

```
SELECT MAX(sal) FROM emp ;
```

```
MAX(SAL)
---------------
5000
```

2.9.2. GROUP BY and HAVING

The GROUP BY operator causes the table represented by the FROM clause to be rearranged into partitions or *groups*, such that within one group all rows have the same value for the GROUP BY column. Note that the table is not physically arranged in the database.

All the SELECT statements we have used until now have acted on data as if the data is in a single group. But the rows of data in some of the tables can be thought of as being part of different groups.

For example, the EMP table contains employees who are CLERKS, SALESMEN etc. If we wish to find the minimum salary of each group of employees, then none of the clauses that we have seen until now are of any use.

The GROUP BY are used to group the result table derived from earlier FROM and WHERE clause. HAVING is used to apply search condition on these groups.

When a GROUP BY clause is used, each row of the resulting table will represent a group having same values in the column(s) used for grouping. HAVING clause then acts on the resulting grouped table to remove the row which does not satisfy the criteria in the HAVING search condition.

The GROUP BY clause is of the form

```
GROUP BY < column  list >
```

The columns specified must be selected in the query. If a table is grouped, the select list should have columns and expressions, which are single-valued for a group. These are

- Columns on which grouping is done
- Constants
- Aggregate functions on the other columns on which no grouping is done

A HAVING clause is of the form

```
HAVING <search condition>
```

The search condition applies to each group. It can be formed using various predicates like (between, in, like, null, comparison) etc. Combined with Boolean (AND, OR, NOT) operators. Since the search condition is for grouped table. The predicates should be

- On a column by which grouping is done.
- a set function (Aggregate function) on other columns.

The aggregate functions can be used in HAVING clause. But they cannot be used in the WHERE clause.

To Find Out Average and Minimum, Maximum salary of each department.

```
Select deptno,AVG(sal),MIN(sal),MAX(sal) FROM emp
       group by deptno;
```

DEPTNO	AVG(SAL)	MIN(SAL)	MAX(SAL)
10	2916.66667	1300	5000
20	2175	800	3000
30	1566.66667	950	2850

Explanation of the above query: Here the table emp is grouped so that one group contains all the rows for dept d1 (say), another contain all rows for dept d2 (say), and so on. The SELECT clause is then applied to each group of the partitioned table (rather than to each row of the original table). Each expression in the SELECT clause must be *single-valued per group*; e.g., it can be (one of) the column(s) named in the GROUP BY clause, or a literal, or an aggregate function such as SUM that operates on al values of a given column within a group and reduces those values to a single value.

To Find Out Average and Maximum, Minimum salary of departments where average salary is greater than 2000

```
SELECT deptno,AVG(sal),MIN(sal),MAX(sal) FROM emp
       GROUP BY deptno HAVING AVG(sal) > 2000;
```

DEPTNO	AVG(SAL)	MIN(SAL)	MAX(SAL)
10	2916.66667	1300	5000
20	2175	800	3000

To list the minimum salary of various categories of employees.

```
SELECT job,MIN(sal) FROM emp
          GROUP BY job;
```

JOB	MIN(SAL)
ANALYST	3000
CLERK	800
MANAGER	2450
PRESIDENT	5000
SALESMAN	1250

Only the column names, which have been used in GROUP BY clause and aggregate columns, can be used in SELECT clause. Grouping can be done on multiple columns also. To find the minimum salary of various categories of employees in various departments, use

```
SELECT deptno,job,MIN(sal) FROM emp
    GROUP BY deptno,job;
```

The output is sorted on the basis of deptno and within each deptno, the output is sorted on the order of job. Reverse the order of deptno and job and see the changes to the output.

To list the minimum salary of various categories of employees, department wise, such that minimum salary is greater than 1500

```
SELECT job,deptno,MIN(sal) FROM emp
    GROUP BY job,deptno HAVING MIN(sal) > 1500;
```

JOB	DEPTNO	MIN(SAL)
ANALYST	20	3000
MANAGER	10	2450
MANAGER	20	2975
MANAGER	30	2850
PRESIDENT	10	5000

To find the minimum salary of MANAGER's in various departments

```
SELECT job, deptno, MIN(sal) FROM emp WHERE job = 'MANAGER'
     GROUP  BY job,deptno;
```

JOB	DEPTNO	MIN(SAL)
MANAGER	10	2450
MANAGER	20	2975
MANAGER	30	2850

It is possible to combine a WHERE clause with a HAVING clause. In above example if you are interested in retrieving data where minimum salary is greater than 2500, use

```
SELECT job, deptno, MIN(sal) FROM emp WHERE job = 'MANAGER'
   GROUP BY  job,deptno  HAVING MIN(sal) >2500 ;
```

2.9.3. ORDER BY Clause

A query with its various clauses (FROM, WHERE, GROUP BY, HAVING) determines the rows to be selected and the columns. The order of rows is not fixed unless an ORDER BY clause is given

An ORDER BY clause is of the form

```
ORDER BY < Sort  list> ASC/DESC
```

The columns to be used for ordering are specified by using the column names or by specifying the serial number of the column in the select list. The sort is done on the column in Ascending or Descending order. The default is Ascending.

Note: GROUP BY does not imply ORDER BY.

To list all employees in the ascending order by name

```
SELECT ename, sal from emp order by ename;

ENAME                SAL
----------           ----------
ADAMS                1100
ALLEN                1600
BLAKE                2850
CLARK                2450
FORD                 3000
JAMES                950
JONES                2975
KING                 5000
MARTIN               1250
MILLER               1300
SCOTT                3000
SMITH                800
TURNER               1500
WARD                 1250
```

To select all employees sorted departmentwise in ascending order and within department, salarywise in descending order.

```
SELECT deptno,ename,sal FROM emp
    order by deptno, sal desc;

DEPTNO  ENAME       SAL
-----   ----------- ----------
10      KING        5000
10      CLARK       2450
10      MILLER      1300
20      SCOTT       3000
20      FORD        3000
20      JONES       2975
20      ADAMS       1100
20      SMITH       800
30      BLAKE       2850
30      ALLEN       1600
30      TURNER      1500
30      WARD        1250
30      MARTIN      1250
30      JAMES       950
```

To select all employees sorted dept wise in ascending order and within dept salary wise in descending order for deptno 10 and 20.

```
SELECT deptno,ename,sal FROM emp
    WHERE deptno=10 or deptno = 20
    ORDER BY deptno,sal DESC;

DEPTNO  ENAME       SAL
------  ------      ----------
10      KING        5000
10      CLARK       2450
10      MILLER      1300
20      SCOTT       3000
20      FORD        3000
20      JONES       2975
20      ADAMS       1100
20      SMITH       800
```

To select all employees along with their annual salary sorted on the basis of the annual salary.

```
SELECT ename, sal * 12  "Annual Salary" FROM emp
   ORDER BY 2 DESC;

Ename          Annual Salary
-----------    ------------------
KING           60000
SCOTT          36000
FORD           36000
JONES          35700
BLAKE          34200
CLARK          29400
ALLEN          19200
TURNER18000
MILLER 15600
WARD           15000
MARTIN15000
ADAMS          13200
JAMES          11400
SMITH          9600
```

2.10. SQL Functions

We have seen some of the aggregate functions provided by ORACLE. ORACLE also provides a rich set of functions, which act on single rows.

If you call a SQL function with a null argument, the SQL function automatically returns null.

Different functions work on different type of data. Following is a list of most commonly used functions from each category.

2.10.1. String functions

1. **UPPER(string)**

- Converts all characters in string to uppercase.

```
SELECT UPPER(ename),empno       FROM emp  WHERE job = 'CLERK';

UPPER(ENAME)             EMPNO
------------ -----------
SMITH                    7902
ADAMS                    7788
JAMES                    7698
MILLER                   7782
```

2. **LOWER(string)**

- Converts all characters in string to lowercase.

```
SELECT LOWER(ename),empno FROM emp WHERE job = 'CLERK';

LOWER(ENAME)             EMPNO
-------------            ----------
smith                    7902
adams                    7788
james                    7698
miller                   7782
```

3. **INITCAP(string)**

Converts the first character of each word in string to uppercase and the rest of the characters to lowercase.

```
SELECT INITCAP(ename),empno FROM emp
WHERE job = 'CLERK';

INITCAP(ENAME)               EMPNO
-----------------            -----------
Smith                        7902
Adams                        7788
James                        7698
Miller                       7782
```

4. **LPAD(string1,n,string2)**

Adds string2 before string1 as many times as required to make the string1 length equal to n chars.
To right align the names of employees

```
SELECT empno,LPAD(ename,8,' ') FROM emp WHERE job='CLERK';
```

```
LPAD(ENAME,8,' ')                EMPNO
-------------------------        -----------
            SMITH                7902
            ADAMS                7788
            JAMES                7698
            MILLER               7782
```

5. **LTRIM(string,CHAR set)**

Removes chars from beginning of string as long as the character matches one of the chars in the CHAR set.

```
SELECT ename,LTRIM(ename,'MALICE') FROM emp;

ENAME                            LTRIM(ENAME,
-----------                      ----------------------
SMITH                            SMITH
ALLEN                            N
WARD                             WARD
JONES                            JONES
MARTIN                           RTIN
BLAKE                            BLAKE
CLARK                            RK
SCOTT                            SCOTT
KING                             KING
TURNER                           TURNER
ADAMS                            DAMS
JAMES                            JAMES
FORD                             FORD
MILLER                           R
14 rows selected.
```

6. **RPAD(string1,n,string2)**

Similar to LPAD. Adds to the right end.

7. **RTRIM(string,CHAR set)**

Similar to LTRIM. Removes at the right end.

8. **SUBSTR(CHAR,m,n)**

This function returns the string from the m th character of the string to the n th character.

```
SELECT SUBSTR('abcdefg',3,2) "SUBSTRING"  from dual;
SUBSTRING
-----------------
cd
```

NOTE: The table named DUAL is a small table in the data dictionary that Oracle and user-written programs can reference to guarantee a known result. This table has one column called DUMMY and one row containing the value X.

2.10.2. Date and Time Functions

1. **SYSDATE**

Returns the current DATE and TIME.

```
SELECT SYSDATE FROM DUAL;
```

2. **TRUNC (DATE1)**

Truncates the time part from the DATE. This is required when we do DATE calculations.

```
SELECT ename, TRUNC(sysdate) - TRUNC(HIREDATE) FROM emp;
```

3. **MONTHS_BETWEEN (DATE1,DATE2)**

Returns number of months between the two DATEs.

```
SELECT empno,MONTHS_BETWEEN(TRUNC(sysdate),hiredate)
     FROM emp;
```

```
EMPNO       MONTHS_BETWEEN(TRUNC(SYSDATE),HIREDATE)
---------  ---------------------------------------------------------------
7369              10.36
7499              1.562
7521              0.721
.                 .
14 rows selected
```

4. **ADD_MONTHS(DATE1,int1)**

Returns a DATE, int1 times added.

```
SELECT ename,ADD_MONTHS(hiredate,2) FROM emp;
```

```
ENAME       ADD_MONTHS(HIREDATE,2)
----------- ----------------------------------------
SMITH             17-FEB-81
ALLEN             20-APR-81
.                       .
.                       .
14 rows selected
```

int1 can also be a negative integer.

5. **TO_CHAR(DATE1,format)**

Converts the DATE given to the format specified

Give below is a list of some of the formats allowed :

Format	Meaning
YYYY	Four Digit Year
YY	Last two digits of the year
MM	Month (01-12;JAN=01…)
MONTH	Name of the month stored as a length of nine characters
MON	Name of the month in three letter format
DD	Day of the month (01-31)
D	Day of the week
DAY	Name of the day stored as a length of nine characters
DY	Name of the day in three letter format
FM	This prefix can be added to suppress blankpadding
HH	Hour of the day
HH24	Hour in the 24 hr format
MI	Minutes of the hour
SS	Seconds of the minute
TH	The suffix used with the day

```
    SELECT TO_CHAR(TRUNC(sysdate), 'ddth fmMonth yy') 'DATE'
       FROM dual;
DATE
---------
01st January 95

    SELECT TO_CHAR(TRUNC(sysdate),'fmMonth') "DATE" FROM dual;
DATE
---------
July
```

6. **ROUND(n,m)**

Returns n rounded to m places.

If m is omitted, to zero places. m can be negative to round off digits left of the decimal point.

```
    SELECT ROUND(17.175,1) "Number" FROM dual;

Number
--------
17.2

    SELECT ROUND(17.175,-1) "Number" FROM dual;

Number
-----------
20
```

7. LASTDAY(d)

Returns the date of the last day of the month that contains d. You might use this function to determine how many days are left in the current month.

```
    SELECT SYSDATE,
       LAST_DAY(SYSDATE) "Last",
       LAST_DAY(SYSDATE) - SYSDATE "Days Left"
       FROM DUAL;

SYSDATE  Last                      Days Left
--------- --------------           ----------
23-OCT-97 31-OCT-97                   8
```

8. NEXT_DAY(d, char)

Returns the date of the first weekday named by char that is later than the date d.

```
    SELECT NEXT_DAY('15-MAR-98','TUESDAY') "NEXT DAY"
         FROM DUAL;

NEXT DAY
---------
16-MAR-98
```

Numeric Functions

9. TRUNC(n,m)

Returns n truncated to m places. If m is omitted, n is truncated to zero places. m can be negative to truncate left of the decimal point.

```
    SELECT TRUNC(15.81,1) "Number"  FROM dual;
Number
----------
15.8
    SELECT TRUNC(15.81,-1) "Number" FROM dual;

Number
-----------
10
```

10. **ROUND(N,M)**

Returns n rounded to m places right of the decimal point.

```
    SELECT ROUND(15.193,1) "Round" FROM DUAL;
Round
----------
      15.2
```

11. **CEIL(n)**

Returns smallest integer greater than or equal to n.

```
    SELECT CEIL(15.7) "Ceiling" FROM DUAL;
Ceiling
----------
        16
```

12. **FLOOR(n)**

Returns largest integer equal to or less than n.

```
    SELECT FLOOR(15.7) "Floor" FROM DUAL;
Floor
----------
        15
```

13. **ABS(n)**

Returns the absolute value of n.

```
    SELECT ABS(-15) "Absolute" FROM DUAL;
 Absolute
----------
        15
```

There are some more numeric functions available, Please refer to the online documentation for more help.

Other SQL Functions are

LENGTH: Returns the length of char in characters

a) LENGTH (string)

```
Select LENGTH ('candide') " length in characters"
From dual;

7 Length in characters
```

b) NVL(exprl,n)

```
NVL substitutes n if value of exprl is NULL

SELECT ename,NVL(COMM,0) "COMMISSION" FROM emp
     WHERE deptno=30;

ENAME       COMMISSION
ALLEN       300
WARD        500
MARTIN      1400
BLAKE       0
TURNER      0
JAMES       0
```

2.10.3. DECODE:

A Special function used to evaluate expression. This works like the IF-THEN-ELSE conditional loop.

```
Decode(col_name,condition1,value1,condition2,value2…..,default value)


Select Ename,job,DECODE(job,'SALESMAN','ELIGIBLE','PRESIDENT','HEAD
              PERSON','NOT ELIGIBLE') "COMM ELIGIBLITY" FROM EMP;

ENAME       JOB             COMM ELIGIBILITY
SMITH       CLERK           NOT ELIGIBLE
SMITH       SALESMAN        ELIGIBLE
WARD        SALESMAN        ELIGIBLE
JONES       MANAGER         NOT ELIGIBLE
MARTIN      SALESMAN        ELIGIBLE
BLAKE       MANAGER         NOT ELIGIBLE
CLARK       MANAGER         NOT ELIGIBLE
SCOTT       ANALYST         NOT ELIGIBLE
KING        PRESIDENT       HEAD PERSON
```

```
TURNER      SALESMAN          ELIGIBLE
ADAMS       CLERK             NOT ELIGIBLE
JAMES       CLERK             NOT ELIGIBLE
FORD        ANALYST           NOT ELIGIBLE
MILLER      CLERK             NOT ELIGIBLE
```

POWER: Returns m raised to nth power ------ power(m,n)

```
POWER(m,n)
SELECT POWER (3,2) "Raised" FROM DUAL;

Raised
------
     9
```

REPLACE: Returns char with every occurrence of search string replaced with replacement string. Refer the online documentation for more detail.

```
a) REPLACE (String, if , then)
   SELECT ename,replace (ename, 'I' , 'x') "changed name" FROM Emp;

ENAME      Changed Name
---------- ------------
KING       KxNG
BLAKE      BLAKE
CLARK      CLARK
JONES      JONES
MARTIN     MARTxN
ALLEN      ALLEN
TURNER     TURNER
JAMES      JAMES
WARD       WARD
FORD       FORD
SMITH      SMxTH
SCOTT      SCOTT
ADAMS      ADAMS
MILLER     MxLLER

b) INSTR(string, pattern[ , start [,occurrence ] ] )
Returns the location of a character IN a STRING
SELECT INSTR ('CORPORATEFLOOR','R',3,2) "Instring" FROM DUAL;

Instring
--------
       6
```

3. Joins and Subqueries

3.1. Joins

The ability to "join" two or more tables is one of the most powerful features of relational systems. In fact, it is the availability of the join operation, almost more than anything else that distinguishes relational from non-relational systems. So what is a join? Loosely speaking, it is *a query in which data is retrieved from more than one table*.

JOINS make it possible to select data from more than one table by means of a single statement.

The joining of tables is done in SQL by specifying the tables to be joined in the FROM clause of the SELECT statement. When you join two tables a Cartesian product is formed. The conditions for selecting rows from the product are determined by the predicates in the WHERE clause. All the subsequent WHERE, GROUP BY, HAVING, ORDER BY clauses work on this product.

If the same table is used more than once in a FROM clause then to resolve conflicts and ambiguities aliases are used. They are also called *co-relation names* or *range variables*. Once aliases are used, table names cannot be used in further clauses.

Assume two tables

```
TABLE F1

COL1    COL2
-----   -----
 A      1
 B      2
 C      3

 Table F2
COL1    COL2
-----   -----
 X      100
 Y      200
```

The statement **SELECT * FROM F1,F2;** results in

```
COL1       COL2       COL1       COL2
---------  ---------- ---------  ---------
  A               1   X          100
  B               2   X          100
  C               3   X          100
  A               1   Y          200
  B               2   Y          200
  C               3   Y          200
```

6 rows selected.

Statement

```
SELECT * FROM F1 First_T, F1 Second_T;
```

results in

```
COL1       COL2    COL1    COL2
---------  ------- -----   ----------
  A            1     A     1
  B            2     A     1
  C            3     A     1
  A            1     B     2
  B            2     B     2
  C            3     B     2
  A            1     C     3
  B            2     C     3
  C            3     C     3
```

9 rows selected.

3.1.1. Equi-join(Inner Join)

Typically the tables are joined to get meaningful data. Suppose we want to get the department name of all employees along with their names. Now since the employee name is in the employee table and dept name is present in dept table we have to take the join of two tables on the basis of the common column between these two tables i.e deptno column.

```
SQL> SELECT ename, dname, sal FROM emp,dept
        WHERE emp.deptno = dept.deptno;

ENAME         DNAME              SAL
-----------   --------------    ----------
SMITH         RESEARCH          800
ALLEN         SALES             1600
WARD          SALES             1250
JONES         RESEARCH          2975
MARTIN        SALES             1250
BLAKE         SALES             2850
CLARK         ACCOUNTING        2450
```

```
SCOTT          RESEARCH          3000
KING           ACCOUNTING        5000
TURNER         SALES             1500
ADAMS          RESEARCH          1100
JAMES          SALES             950
FORD           RESEARCH          3000
MILLER         ACCOUNTING        1300
14 rows selected.
```

The join is based on the equality of column values in the two tables (emp.deptno= dept.deptno) and therefore is called an *equi-join*.

The equi-join is also used to provide summary information. Suppose we wanted to know the details of all departments along with number of employees working in it. Since the number of employees can be found out only through the emp table we have to take a join. Also since we want number of employees per dept, we have to group the product of the join.

```
SQL> SELECT dept.deptno, dept.dname, dept.loc, count(*) "NO of
          employees"  FROM dept,emp
          WHERE emp.deptno=dept.deptno
          GROUP BY dept.deptno,dept.dname,dept.loc;

DEPTNO     DNAME          LOC            NO of Employees
---------- -------------- ---------- ----------------------
10         ACCOUNTING     NEW YORK          3
20         RESEARCH       DALLAS            5
30         SALES          CHICAGO           6
```

NULL is an unknown and not a value, NULL values never satisfy equi-join condition.

3.1.2. NON EQUI-JOINS (Theta Join)

When the comparison operator used in joining columns is other than equality, the join is called a *Non Equi-join*.

Suppose we want to find the grades of employees based on their salary

```
SQL> select ename,grade,sal from emp,salgrade where sal between
       losal and hisal;
```

```
ENAME           GRADE       SAL
---------------------------------
SMITH               1       800
ADAMS               1      1100
JAMES               1       950
WARD                2      1250
MARTIN              2      1250
MILLER              2      1300
ALLEN               3      1600
TURNER              3      1500
JONES               4      2975
BLAKE               4      2850
CLARK               4      2450
SCOTT               4      3000
FORD                4      3000
KING                5      5000

14 rows selected
```

3.1.3. SELF JOIN

In a self-join a Cartesian product is taken on two sets of rows of the same table and a condition is applied on these rows.

.

```
TABLE  F1
COL1          COL2
-----   -----
 A             1
 B             2
 C             3

SELECT * FROM f1 first_f1, f1  second_f1
         WHERE first_f1.col1 = second_f1.col1;

COL1           COL2       COL1        COL2
---------      --------   ---------   ---------
A               1          A          1
B               2          B          2
C               3          C          3
```

In the above query the join is taken between the rows of the same table so it also becomes a self-join.

To get the names and salaries of all employees who have managers along with manager's name and managers salary.

```
SQL> SELECT a.ename  name ,  a.sal salary , b.ename mgr, b.sal " Mgr
        Salary" FROM emp a , emp b   WHERE  A.mgr = B.empno;
```

```
NAME        SALARY      MGR         Mgr Salary
----------  -------     --------    ---------------
SMITH         800       FORD            3000
ALLEN        1600       BLAKE           2850
WARD          1250      BLAKE           2850
JONES         2975      KING            5000
MARTIN        1250      BLAKE           2850
BLAKE         2850      KING            5000
CLARK         2450      KING            5000
SCOTT         3000      JONES           2975
TURNER       1500       BLAKE           2850
ADAMS        1100       SCOTT           3000
JAMES         950       BLAKE           2850
FORD          3000      JONES           2975
MILLER        1300      CLARK           2450
13 rows selected.
```

*** King is not selected because he does not have a manager**

3.1.4. OUTER JOINS

Consider the query for getting all the department details along with the number of employees in it. In an equi-join, the details of deptno 40 were not shown because there are no employees assigned to it in the EMP table.

This can be avoided by taking an outer join. Outer Join is an exclusive union of sets (whereas normal joins are intersection). Outer Joins can be simulated using UNIONS.

In a join of two tables an outer join may be for the first table or the second table. If the outer join is taken on say the dept table then each row of dept table will be selected at least once whether or not a join condition is satisfied.
The general syntax is

```
WHERE  table1 < OUTER JOIN INDICATOR >  =  table 2
```

In Oracle it is (+)

```
table1.column = table2.column(+)  means OUTER join is taken on
table1.
```

- To get the details of all departments with number of employees assigned to it (even 0 employees)

```
SELECT dept.deptno, dept.dname, dept.loc, count(empno) "No. of
employees" FROM dept, emp
```

```
    WHERE emp.deptno(+)=dept.deptno
    GROUP BY dept.deptno,dept.dname,dept.loc;

DEPTNO     DNAME              LOC           No.  of  employees
-----------------------     ---------    --------------------
10         ACCOUNTING         NEW YORK            3
20         RESEARCH           DALLAS              5
30         SALES              CHICAGO             6
40         OPERATIONS         BOSTON              0
```

Suppose there were any employees without deptno assigned to it. If we take the equi-join then employees without any dept will not be selected because join condition will not be satisfied for NULL values. To solve this problem we have to take an outer join on the department table.

The query to get all the names, salary, department names of all employees we have to take an outer join on employee table.

```
SELECT ename, sal,dname  FROM emp,dept WHERE
dept.deptno(+)=emp.deptno;
```

3.2. Subqueries

When the WHERE clause needs a set of values which can be only obtained from another query, the subquery is used. In the WHERE clause it can become a part of the following predicates

- COMPARISON Predicate
- IN Predicate
- ANY or ALL Predicate
- EXISTS Predicate

It can be also used as a part of the condition in the HAVING clause.
To list the names of all employees working in SALES dept, assuming that deptno of SALES dept is not known.

```
SELECT ename,sal FROM emp
  WHERE deptno = (SELECT deptno
                    FROM dept
                    WHERE dname = 'SALES');

ENAME                SAL
-----------          ----------

ALLEN               1600
WARD                1250
MARTIN              1250
```

```
BLAKE                2850
TURNER               1500
JAMES                950
```

The first select statement is called as OUTER QUERY or MAIN QUERY. The second SELECT statement is called INNER QUERY or SUB-QUERY. All the rules, which apply to the main query, also apply to the sub-query. The inner query is executed FIRST and ONLY ONCE. The inner query returns a single value, namely 30. This value is used to evaluate the main query. Notice that the table used in inner query can be different from the table used in outer query.

- The sub-query cannot have an ORDER BY clause and UNION of SELECT's
- The sub-query may select only one column except in the case of EXISTS predicate

There are two types of sub-queries – Co-related and Non Co-related sub-queries. If the sub-query uses any data from the FROM clause of the outer query, the sub-query is evaluated for each row of the outer query. The sub-query thus has to be repeated for each row of the outer query and is called *co-related sub-query*.

When no data is needed from the outer query, the sub-query is evaluated only once. The sub-query is executed and the result set is obtained. Such a sub-query is called as non co-related sub-query.

3.2.1. SUB-QUERIES IN PREDICATES

Comparison

<EXPRESSION> < OPERATOR > < SUBQUERY>

The sub-query should result in one value of the same data type as the left-hand side.
To get all the employees whose salary is greater than the average salary of the company.

```
SQL> SELECT ename,sal FROM emp
 WHERE sal > ( SELECT AVG(sal) FROM emp);
```

```
ENAME          SAL
----------  ----------
CLARK            2450
BLAKE            2850
JONES            2975
SCOTT            3000
FORD             3000
KING             5000
```

To select all employees who do the same job as that of 'SCOTT'.

```
SQL> SELECT  empno,ename, sal FROM  emp
2 WHERE JOB=(SELECT job FROM emp WHERE ename='SCOTT');

EMPNO       ENAME       SAL
----------  ----------  ----
7788        SCOTT       3000
```

3.2.2. IN PREDICATE IN SUB-QUERIES

<EXPRESSION> IN < SUBQUERY>

The list of values generated by the sub-query should be of same datatype as the left-hand side.

To see all details of departments who have employees working in it.

```
SQL> SELECT  * FROM dept
WHERE deptno IN (SELECT distinct deptno FROM emp);

DEPTNO      DNAME       LOC
----------  ----------  -------------
10          ACCOUNTING  NEW YORK
20          RESEARCH    DALLAS
30          SALES       CHICAGO
```

3.2.3. ALL, ANY PREDICATES

These predicates are called *Quantified Predicates*. They allow testing of a single value against all members of the set.

The general form is

< EXPRESSION> <OPERATOR> < ANY /ALL/SOME > <SUB-QUERY>

The datatypes must be comparable to the left and right hand side. The sub-query generates a list of values of the same type as the <EXPRESSION> on the left and the operator is any comparison operator.

FOR ALL the predicate is evaluated as follows:

1. True if the comparison is true for every value of the list of values.
2. True if sub-query gives a null set (No values)
3. False if the comparison is false for one or more of the list of values generated by the sub-query.

FOR ANY the predicate is evaluated as follows:

1. True if the comparison is true for one or more values generated by the sub-query.
2. False if sub-query gives a null set (No values)

3. False if the comparison is false for every value of the list of values generated by the sub-query.

Examples of ANY, ALL

To find names and salary details of all employees who earn more than the managers.

```
SQL> SELECT ename,sal FROM emp
2 WHERE sal > ANY ( SELECT sal FROM emp WHERE JOB='MANAGER') AND JOB
!='MANAGER';

ENAME             SAL JOB
---------- --------- ---------
SCOTT            3000 ANALYST
KING             5000 PRESIDENT
FORD             3000 ANALYST

3 rows selected.
```

To select the employees whose sal is greater than the sal of all the employees working in deptno 30.

```
SQL> SELECT ename,sal FROM emp
2 WHERE sal > ALL (  SELECT sal FROM emp WHERE deptno=30);

ENAME         SAL
------------ ----------
JONES         2975
SCOTT         3000
KING          5000
FORD          3000
```

3.2.4. Co- Related Sub-queries

In the inner query one may refer to the identifiers available from the outer query with proper identification.

```
SQL> SELECT ename, mgr, hiredate FROM emp a
 2 WHERE a.hiredate < (SELECT b.hiredate FROM emp b
                         WHERE b.empno=a.mgr);

ENAME         MGR        HIREDATE
----------    -------    ----------------
SMITH         7902       17-DEC-80
ALLEN         7698       20-FEB-81
WARD          7698       22-FEB-81
JONES         7839       02-APR-81
BLAKE         7839       01-MAY-81
CLARK         7839       09-JUN-81
6 rows selected.
```

Here two sets of rows of EMP table will be created one for the outer query and one for the inner query. In the first pass all the rows of the second set will be compared with first row of first set. This process will be repeated for all the rows of the first set. If for each comparison the condition becomes true then the row from the first set will be selected.

```
SELECT ename, sal, job from emp a
    Where sal > (select avg (sal) from emp b where b.job = a.job);


ENAME       SAL JOB
------      ---------

SMITH       1600 SALESMAN
JONES       2975 MANAGER
BLAKE       2850 MANAGER
TURNER      1500 SALESMAN
MILLER      1300 CLERK
```

3.2.5. EXISTS Predicate

The EXISTS Predicate is of the form

```
<Query> WHERE
EXISTS < sub-query>
```

Query will be evaluated if EXISTS takes a value TRUE. It takes the Value TRUE if the <sub-query> result table is non null and FALSE if it is Null. It is mostly used with Co-Related Subqueries. **EXISTS does not check for a particular value. It checks whether subquery returns rows or not which match to outer query rows.**

To list all details of dept which has at east one employee assigned to it.

```
SQL> SELECT * FROM dept a
2   WHERE EXISTS  (SELECT * FROM EMP b
                    WHERE b.deptno = a.deptno);

DEPTNO  DNAME          LOC
------  -------------- -------------
10      ACCOUNTING     NEW YORK
20      RESEARCH       DALLAS
30      SALES          CHICAGO
```

Here there are 4 rows in dept table so the Inner Query will be executed 4 times for each row returned by outer query. Since there are no employees for deptno 40 the sub-query will return no rows i.e. a null set. Hence EXISTS will become false and the outer query will not select the row in dept table with value 40.

4. Advanced SQL

4.1. Index

Index is a database object, which speeds up access to the data. There are different types of index, which the user can create.

```
CREATE [UNIQUE] INDEX index_name
    ON table_name(col_name1 [ASC|DESC],col_name2,.....);
```

- A table can have any number of indices
- An index can be on a single column or on multiple columns. In 8i, Up to 32 columns can be included in one index
- Updation of all indices is handled by Oracle
- UNIQUE ensures that all values in an index are unique
- Index is ascending by default
- ORACLE decides when to use the index while accessing the data
- Removal of an index does not affect the table on which it is based
- When you create a primary or a unique constraint on the table, a unique index is automatically created for you. The name of the constraint will be used for the index name

Example :

```
CREATE INDEX  emp_sal_index ON emp(sal);
```

- To allow only unique values in ename field, the create statement will be,

```
CREATE UNIQUE INDEX  emp_ename_unindex ON emp( ename );
```

4.2. Synonym

Synonyms are objects, which contain a duplicate name of an existing object.

```
CREATE [PUBLIC] SYNONYM another_name
       FOR existing_name
```

- Existing_name can be name of a table or view or sequence
- PUBLIC is used to grant permission to all users to access the object using the new name. (Only by DBA)
- Useful in hiding ownership details of an object

```
CREATE PUBLIC SYNONYM new_emp FOR emp;
```

4.3. Sequence

Sequence is an object, which can be used to generate sequential numbers. The numbers, which are generated using a sequence, are usually used to fill up columns that are declared as UNIQUE or PRIMARY KEY columns.

```
CREATE SEQUENCE seq_name
        [INCREMENT BY n1]
        [START WITH n2]
        [MAXVALUE n3]
        [MINVALUE n4]
        [CYCLE|NOCYCLE];
```

- START WITH indicates the first number in the series
- INCREMENT BY is the difference between consecutive numbers. If n1 is negative, numbers that are generated are descending in order
- MAXVALUE & MINVALUE indicate the extreme values
- CYCLE indicates that once the extreme is reached, start the cycle again with n2
- NOCYCLE means that once the extreme is reached stop generating numbers

```
CREATE SEQUENCE  s1
    INCREMENT BY 1
    START WITH 1
    MAXVALUE 10000
    NOCYCLE ;
```

s1 will generate NUMBERs 1,2,3....10000 and stop.

The NUMBER generated by s1 can be used to fill empno column of emp table.

4.4. View

View is a logical table based on one or more tables.

```
CREATE VIEW view_name [alias]
  AS query
[WITH CHECK OPTION];
```

- View can be used as if it is a table
- View does not contain data
- Whenever a view is accessed, the query is evaluated. Thus view is dynamic
- Any changes made in the view affect the tables on which the view is based

- View helps to hide the ownership details of a table and complexity of query used to retrieve data, from the user
- "With check option" means you cannot insert a row in the underlying table, which cannot be selected using the view
- If you use a order by clause in the view, then it becomes a read only view automatically

```
CREATE VIEW newempview
  AS
   SELECT * FROM emp
     WHERE hiredate >'01-jan-82';
```

Read Only View

- If a view is based on a single table the user may manipulate records in the view and its underlying base table. The WITH READ ONLY clause of the CREATE VIEW command can be used to prevent users from manipulating records in a view.

There are certain restrictions on queries, which can be used to create a view. The restrictions will be explained when queries are dealt with.

Updation /Insertion of rows in the base table using views is possible only if the view is based on a single table with some restrictions. The restrictions are:

1. Updation /Insertion Not possible if view is based on two tables. Can be done in ORACLE 8 by using INSTEAD OF trigger.
2. Insertion is not allowed if the underlying table has any **not NULL columns** that **do not appear** in the view.
3. Insertion/Updation is not allowed if any column of the view referenced in update/Insert contains functions or calculations.
4. Insertion/Updation /deletion not allowed if view contains group by or distinct clause in the query.

5. PL/SQL

5.1. Introduction to PL/SQL

Procedural Language (PL/SQL) is a procedural extension to SQL, provided by ORACLE. Through PL/SQL, the data manipulation capabilities of SQL are combined with the processing capabilities of a procedural language. (Pro * C, Pro * COBOL).

PL/SQL provides features like conditional execution, looping and branching. PL/SQL supports subroutines too.

The basic unit in any PL/SQL program is a block, which can occur sequentially (one after the other) or nested (One inside the other).

Anonymous blocks are generally constructed dynamically and executed only once.

Named blocks are anonymous blocks with a label that gives the block a name. They are also constructed dynamically and executed only once.

Subprograms are procedures, packages and functions that are stored in the database. These blocks generally don't change once they are constructed and are executed many times by calling them explicitly via a call to a procedure, package or function.

TRIGGERS are named blocks that are stored in the database. They are executed implicitly whenever the triggering event occurs. The triggering event is a DML statement executed against a table in the database.

5.2. PL/SQL block and its Sections

A PL/SQL block is a set of SQL statements, which collectively accomplish a particular task. SQLPLUS commands cannot be used in a PL/SQL block. DDL commands such as CREATE and ALTER cannot be used in a PL/SQL block. All DML and DCL statements can be used in PL/SQL though some modification is required in the syntax of some of the commands.

A PL/SQL block is called so, as the statements are written in a block-structured way as in PASCAL. Every block has a keyword, which indicates the

beginning of a block, and optionally a keyword, which indicates the end of a block.

SAMPLE PL/SQL BLOCK

```
DECLARE                                    -- Declaration Section
   V_Salary                  NUMBER(7,2);
   /* V_Salary is a variable declared in a PL/SQL block. This variable
   is used to store JONES' salary. */
BEGIN                                 -- Execution Section
   SELECT sal INTO V_Salary
   FROM emp WHERE ename = 'JONES'
  EXCEPTION                                -- Exception Section

  WHEN no_data_found THEN
    DBMS_OUTPUT.PUT_LINE('Employee Jones does not exist');
END ;                           -- End of Block
/                               -- PL/SQL block terminator


Output
SQL> /
PL/SQL procedure successfully completed.
```

Example 5.1

Notations are as follows:

1. -- is a single line comment.
2. /* */ is a multi-line comment.
3. Every statement must be terminated by a semicolon.
4. PL/SQL block is terminated by a slash / on a line by itself

A PL/SQL block must have a Declaration section and an Execution section. It can optionally have an Exception section too.

Declaration section is where all the variables and cursors used in a block are declared. Execution section has SQL statements and PL/SQL constructs which aid in BRANCHING, LOOPING and CONDITIONAL EXECUTION. Exception section has statements, which handle the exceptions.

5.3. Declaration Section

Declaration section is used to declare variables, constants, exceptions and cursors. PL/SQL supports all ANSI SQL datatypes.

5.3.1. Variable Declaration

Communication with the database takes place via variables in the PL/SQL block.
The general syntax for declaring a variable is

```
Variable_Name  TYPE  [CONSTANT] [DEFAULT] [ NOT NULL]   [:= Value];
```

```
DECLARE
    V_Description       VARCHAR2(25) ;
    V_Sal               NUMBER(5)  NOT NULL  := 3000 ;
    V_Compcode          VARCHAR2(20)  CONSTANT := 'ABC CONSULTANTS' ;
    V_Comm              NOT NULL DEFAULT   0 ;
```

If a variable is not initialized, it is assigned a NULL value by default. If a NOT NULL is present in the declaration then the variable must be initialized. You cannot assign a null value to a NOT NULL variable in the executable or exception section.

If a CONSTANT is present in the declaration, the variable must be initialized and its value cannot be changed from its initial value.

5.3.2. PL/SQL Datatypes

All PL/SQL datatypes are classified as scalar, reference and composite type. Scalar datatypes do not have any components within it, while composite datatypes have other datatypes within them. A reference datatype is a pointer to another datatype.

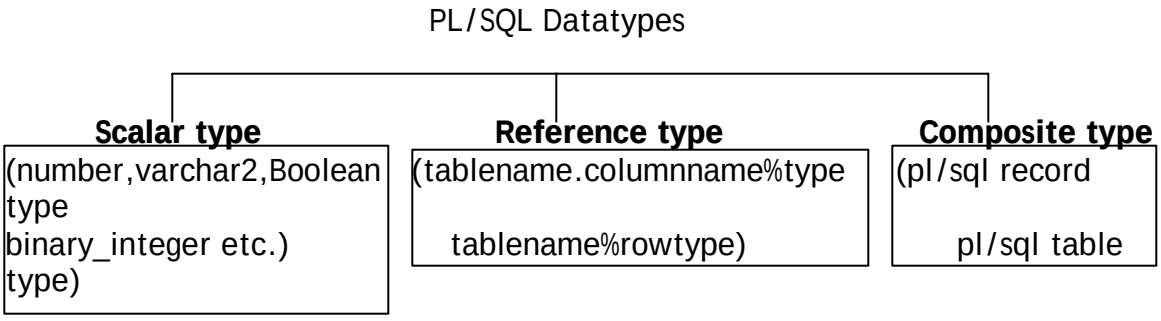


Figure 5-1 Datatypes

5.3.3. SCALAR TYPES

1. NUMBER

This can hold a numeric value, either integer or floating point. It is same as the number database type.

2. BINARY_INTEGER

   If a numeric value is not to be stored in the database, the BINARY_INTEGER datatype can be used. It can only store integers from -2147483647 to + 2147483647. Mostly used for counter variables.

```
V_Counter BINARY_INTEGER DEFAULT 0 ;
```

3. VARCHAR2(L)

   L is necessary and is max length of the variable. This behaves like VARCHAR2 database type. The maximum length in PL/SQL is 32,767 bytes where as VARCHAR2 database type can hold max 2000 bytes. If a VARCHAR2 PL/SQL column is more than 2000 bytes, it can only be inserted into a database column of type LONG .

4. CHAR(L)

   Here L is the maximum length. Specifying length is optional. If not specified it defaults to 1. The maximum length of CHAR PL/SQL variable is 32,767 bytes whereas that of database CHAR column is 255 bytes. Therefore a CHAR variable of more than 255 bytes can be inserted in the database column of VARCHAR2 or LONG type.

5. LONG

   PL/SQL LONG type is just 32,767 bytes. It behaves similar to LONG DATABASE type.

6. DATE

   The DATE PL/SQL type behaves the same way as the date database type. The DATE type is used to store both date and time. A DATE variable is 7 bytes in PL/SQL.

7. BOOLEAN

   A Boolean type variable can only have one of 2 values either TRUE or FALSE. They are mostly used in control structures.

```
V_Does_Dept_Exist  BOOLEAN  := TRUE ;
V_ Flag  BOOLEAN := 0 ;     -- illegal
```

5.3.4. REFERENCE TYPES

A reference type in PL/SQL is the same as a pointer in C. A reference type variable can point to different storage locations over the life of the program.

USING %TYPE

%TYPE is used to declare a variable with the same datatype as a column of a specific table.

```
Var_Name                    table_name.col_name%TYPE ;
V_Empno                     emp.empno%TYPE ;
```

Datatype of V_Empno is same as that of Empno column of the EMP table.

USING %ROWTYPE

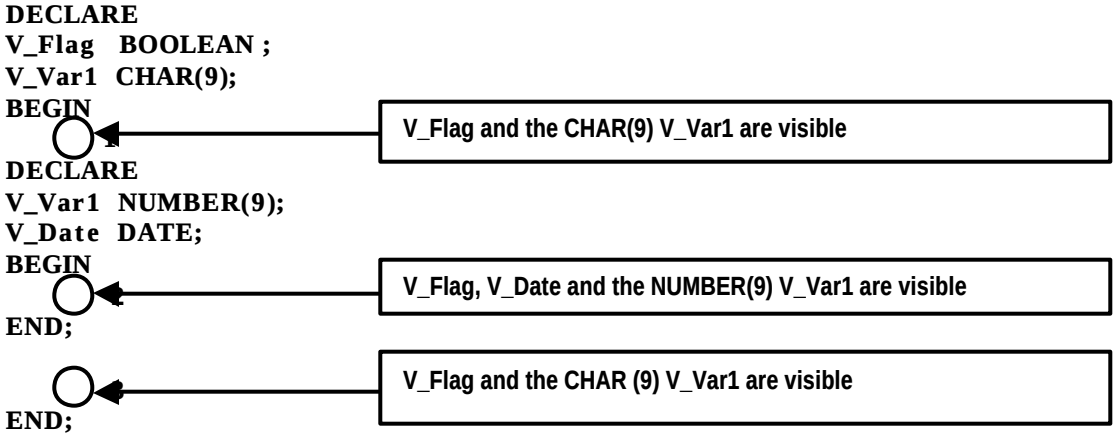
%ROWTYPE is used to declare a compound variable whose type is same as that of a row of a table.

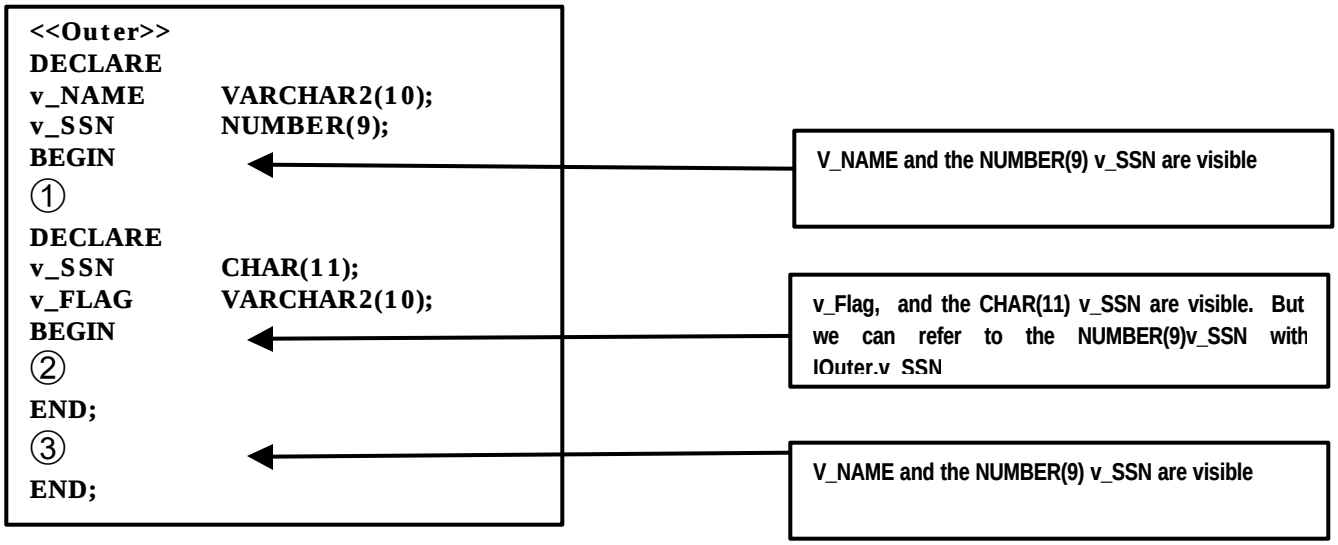
```
Var_Name                    table_name%ROWTYPE ;
V_Emprec                    emp%ROWTYPE ;
```

V_Emprec is a variable, which contains within itself as many variables whose names and datatypes match those of the EMP table. To access the Empno element of V_Emprec, use

V_Emprec.empno ;

5.4. Variable Scope And Visibility





SQL IN PL/SQL

SELECT STATEMENT

```
SELECT Column_List INTO Variable_List
   FROM Table_List
   [WHERE expr1]
   [CONNECT BY expr2 [START WITH expr3]]
   [GROUP BY expr4] [HAVING expr5]
   [UNION | INTERSECT | MINUS SELECT ...]
   [ORDER BY expr | ASC | DESC]
   [FOR UPDATE [OF Col1,...] [NOWAIT]]
   INTO Variable_List
```

The column values returned by the SELECT command must be stored in variables. The Variable_List should match Column_List in both count and datatype. Here the variable_list is PL/SQL (Host) variables, which should be defined before use.

This clause is used if the selected row must be modified through a DELETE or UPDATE command. Since the contents of the row must not be modified between the SELECT command and the UPDATE command, it is necessary to lock the row after the SELECT command. FOR UPDATE will lock the selected row. OF Col1, lists the columns, which can be modified by the UPDATE command. If OF Col1,... is missing, all the columns can be modified.

It is possible that when SELECT..UPDATE is issued, the concerned row is already locked by some other user or a previous SELECT..UPDATE command. If NOWAIT is not given, then the current SELECT..UPDATE command will wait until the lock

is cleared. IF NOWAIT is given, then the SELECT…UPDATE will return immediately with a failure.

```
DECLARE
  V_Emprec      emp%ROWTYPE;
  V_Ename       emp.ename%TYPE;
  V_Sal         emp.sal%TYPE;
BEGIN
  SELECT * INTO V_Emprec FROM emp WHERE ename ='SMITH';
  SELECT ename,sal INTO V_Ename,V_Sal from emp where ename ='SMITH';
END;
```

Example 5.2

INSERT STATEMENT

The syntax for the insert statement remains the same as in SQL –INSERT.

```
DECLARE
     V_Dname   VARCHAR2(15) := 'FINANCE';
BEGIN
     INSERT INTO  dept VALUES ( 50 , V_Dname, 'BOMBAY');
END;
/
```

Example 5.3

DELETE STATEMENT

DELETE FROM table_name
WHERE [CURRENT OF Cursor_Name|Condition]

```
DECLARE
     V_Sal_Cutoff  NUMBER(4):= 2000;
BEGIN
     DELETE FROM emp  WHERE  sal < V_Sal_Cutoff;
END;
/
```

Example 5.4

UPDATE STATEMENT

UPDATE table_name SET col_name = Value
WHERE [CURRENT OF Cursor_Name|Condition]

```
DECLARE
     V_Sal_Incr  NUMBER(5) := 1000;
BEGIN
     UPDATE emp SET sal = sal + V_Sal_Incr WHERE ename='SMITH';
END;
/
```

Example 5.5

VARIABLE NAMES

Consider the block

```
DECLARE
     V_Deptno     NUMBER(10)  := 30;
     V_Dname            VARCHAR2(15) ;
BEGIN
     SELECT dname INTO  V_Dname  FROM dept WHERE deptno =  V_Deptno;
END;
```

Example 5.6

Here there is no ambiguity between the database column names and PL/SQL variables.

```
DECLARE
     Deptno       NUMBER(10)  := 30;
     V_dname      VARCHAR2(15) ;
BEGIN
     SELECT dname INTO V_dname FROM dept WHERE deptno =   deptno;
     DELETE FROM dept WHERE deptno = deptno ;
END;
```

Example 5.7

Here the select statement will select names of all the departments and not only deptno. 30. The delete statement will delete all the employees. This happens because when the PL/SQL engine sees a condition expr1 = expr2, expr1 and expr2 are first checked to see whether they match the database columns first and then the PL/SQL variables. So in above example deptno = deptno both are treated as database columns and the condition will become true for every row of the table.

If a block has a label then variables with same names as database columns can be used by using <<blockname>>. Variable_Name notation.

```
<<BLOCK1>>
DECLARE
     Deptno   NUMBER(10)  := 30;
     Dname   VARCHAR2(15) ;
BEGIN
     SELECT Dname INTO dname FROM dept WHERE deptno =   Block1. deptno;
     DELETE FROM dept WHERE deptno = Block1. deptno ;
END;
```

Example 5.8

Although this method can be used to get the desired results, it is not a good programming practice to use same names for PL/SQL variables and database columns.

5.5. Programming Constructs

5.5.1. LOOPING

A LOOP is used to execute a set of statements more than once. The syntax of a LOOP statement is

```
LOOP
  PL/SQL_Statements;
END LOOP ;
```

```
DECLARE
      V_Counter  NUMBER := 50 ;
BEGIN
      LOOP
         INSERT INTO dept
               VALUES(V_Counter,'NEW DEPT','SOMEWHERE');
         V_Counter := V_Counter + 10 ;
      END LOOP;
      COMMIT ;
END ;
/
```

Example 5.13

This is an endless loop. When LOOP ENDLOOP is used in the above format, then an exit path must necessarily be provided. Exit path is provided using EXIT or EXIT WHEN commands.

EXIT is an unconditional exit. Control is transferred to the statement following END LOOP, when the execution flow reaches the EXIT statement.

```
BEGIN
.....
LOOP
IF <Condition> THEN
. . . . .
EXIT ;                  -- Exits loop immediately
END IF ;
END LOOP;
. . . . .               -- Control resumes here
COMMIT ;
END ;
/
```

EXIT WHEN is for conditional exit out of the loop.

```
DECLARE
       V_Counter        NUMBER := 50 ;
  BEGIN
       LOOP
          INSERT INTO dept
      VALUES(V_Counter,'NEWDEPT','SOMEWHERE') ;
          DELETE FROM emp     WHERE deptno = V_Counter ;
          V_Counter := V_Counter + 10 ;
          EXIT WHEN  V_Counter >100 ;
       END LOOP;
       COMMIT ;
  END ;
/
```

Example 5.14

As long as V_Counter has a value less than or equal to 100, the loop continues.

LOOP…END LOOP can be used in conjunction with FOR and WHILE for better control on looping.
The syntax of FOR LOOP is

```
FOR Variable IN [REVERSE] Lower_Bound..Upper_Bound
LOOP
     PL/SQL_Statements
END LOOP ;
```

FOR loop is used for executing the loop a fixed number of times. The number of times the loop will execute equals
Upper_Bound - Lower_Bound + 1.
Upper_Bound and Lower_Bound must be integers and Upper_Bound must be equal to or greater than Lower_Bound. Variables in FOR loop need not be explicitly declared. Variables take values starting at a Lower_Bound and ending at a Upper_Bound. The variable value is incremented by 1, every time the loop reaches the bottom. When the variable value becomes equal to the Upper_Bound then the loop executes and exits.

When REVERSE is used, then the variable takes values starting at Upper_Bound and ending at Lower_Bound. Value of the variable is decremented each time the loop reaches the bottom.

```
DECLARE
V_Counter   NUMBER := 50 ;
BEGIN
FOR  Loop_Counter  IN 2..5
LOOP
INSERT INTO dept
VALUES(V_Counter,'NEW DEPT','SOMEWHERE') ;
V_Counter := V_Counter + 10 ;
END LOOP;
COMMIT ;
END ;
```

Example 5.15

The loop will be executed (5 - 2 + 1) = 4 times. A Loop_Counter variable can also be used inside the loop, if need be. Lower_Bound and/or Upper_Bound can be integer expressions also.

The syntax for the WHILE LOOP is

WHILE Condition
LOOP
    PL/SQL Statements;
END LOOP;

EXIT OR EXIT WHEN can be used inside the while loop to exit the loop prematurely.

5.5.2. LABELING LOOPS

Loops themselves can be labeled as in the case of blocks. The label can be used with the EXIT statement to exit out of a particular loop

```
BEGIN
     <<Outer_Loop>>
     LOOP
       PL/SQL
       << Inner_Loop>>
       LOOP
            PL/SQL  Statements ;
            EXIT Outer_Loop WHEN <Condition Met>;
       END LOOP  Inner_Loop;
     END LOOP  Outer_Loop;
END ;
 /
```

Example 5.16

5.5.3. CONDITIONAL EXECUTION

Conditional Execution construct is used to execute a set of statements only if a particular condition is TRUE or FALSE.

```
IF Condition_Expr THEN
   PL/SQL_Statements
End if;
```

When the condition evaluates to TRUE then the PL/SQL statements are executed, otherwise the statement following END IF is executed.

```
IF V_Empno = 7902 THEN
   UPDATE emp
   SET sal = sal + 100 WHERE empno = 7902;
END IF;
```

UPDATE statement is executed only if value of V_Empno variable equals 7902.

PL/SQL allows many variations of the IF – END IF construct.

To take alternate action if the condition is FALSE, use

```
IF Condition_Expr THEN
  PL/SQL_Statements_1 ;
ELSE
   PL/SQL_Statements_2 ;
END IF;
```

When the condition evaluates to TRUE PL/SQL_Statements_1 is executed, otherwise PL/SQL_Statements_2 is executed.

The above syntax checks only one Condition_Expr.

To check for multiple conditions and take action based on the condition, use

```
IF Condition_Expr_1 THEN
  PL/SQL_Statements_1 ;
ELSIF Condition_Expr_2 THEN
  PL/SQL_Statements_2 ;
ELSIF Condition_Expr_3 THEN
  PL/SQL_Statements_3 ;
ELSE
  PL/SQL_Statements_n ;
END IF;
```

As every condition must have at least one statement, NULL statement can be used as filler. NULL command does nothing. Sometimes NULL is used in a condition merely to indicate that such a condition has also been taken into consideration. So your code will resemble the code as given below:

```
IF Condition_Expr_1 THEN
     PL/SQL_Statements_1 ;
ELSIF Condition_Expr_2 THEN
     PL/SQL_Statements_2 ;
ELSIF Condition_Expr_3 THEN
      Null;
END IF;
```

CONDITIONS FOR NULL are checked through IS NULL and IS NOT NULL predicates.

6. Error Handling

A good programming language should provide capabilities of handling errors and recovering from them if possible. PL/SQL implements error handling via exceptions and exception handlers.

6.1. Types of Errors in PL/SQL

Compile Time errors are reported by the PL/SQL compiler and you have to correct them before recompiling.
The run-time engine reports run time errors and they are handled programmatically by raising an exception and catching it in the Exception section

6.2. Declaring Exceptions

Exceptions are declared in the Declaration section, raised in the executable section and handled in the Exception Section. There are two types of exceptions predefined and user defined.

6.2.1. User Defined Exceptions

It is an error defined by the program. It could be an error with the data.

```
DECLARE
     E_Balance_Not_Sufficient EXCEPTION;
     E_Comm_Too_Large EXCEPTION;
     …
BEGIN
     NULL;
END;
```

6.2.2. Predefined Exceptions

ORACLE has predefined several exceptions that correspond to the most common ORACLE errors. They are always available to the program; hence there is no need to declare them. These exceptions are automatically raised by ORACLE whenever that particular error condition occurs.

Some of the Predefined Exceptions

NO_DATA_FOUND

This is raised when SELECT INTO …. Statement does not return any rows.

TOO_MANY_ROWS

This is raised when SELECT INTO …. Statement returns more than one row.

INVALID_CURSOR

This is raised when an illegal cursor operation is performed such as closing an already closed cursor.

VALUE_ERROR

This exception is raised when an arithmetic, conversion, truncation or constraint error occurs in a procedural statement.

DUP_VAL_ON_INDEX

This is raised when the UNIQUE CONSTRAINT is violated.

Raising Exceptions

When the error associated with an exception occurs, the exception is raised. This is done through the RAISE command. The syntax is

RAISE Exception_Name ;

An exception is defined and raised as shown below:

```
DECLARE
     ...
     Retired_Emp EXCEPTION ;
BEGIN
     PL/SQL_Statements ;
     IF Error_Condition THEN
          RAISE Retired_Emp ;
          PL/SQL_Statements ;
        END IF;
EXCEPTION
     WHEN Retired_Emp THEN
     PL/SQL_Statements ;
END ;
```

Control passing to Exception Handler

```
DECLARE
   Dup_Deptno  EXCEPTION;
   V_Counter BINARY_INTEGER;
   V_Department NUMBER(2) := 50;
BEGIN
  SELECT COUNT(*) INTO V_Counter  FROM Dept WHERE  deptno = 50;
  IF V_Counter > 0 THEN
    RAISE Dup_Deptno ;
  END IF;
  INSERT INTO dept VALUES (V_Department ,'NEW NAME', 'NEW LOC');
EXCEPTION
  WHEN Dup_Deptno THEN
  INSERT INTO Error_Log VALUES ('Department ' || V_Department
                                || '  IS ALREADY PRESENT');
END ;
/
```

Example 6.1

OTHERS exception handler

The OTHERS handler will execute for all raised exceptions. It should always be the last handler in the block. The OTHERS exception handler handles all errors, which are not handled separately in the EXCEPTION section. To handle a specific case within the OTHERS handler, predefined functions SQLCODE and SQLERRM are used.

SQLCODE returns the current error code. And SQLERRM returns the current error message text. The values of SQLCODE and SQLERRM should be assigned to local variables before using it within a SQL statement.

```
DECLARE
     Err_Num NUMBER ;
     Err_Msg CHAR(100);
BEGIN
      -------
      ------
EXCEPTION
     ----------
     -------
     WHEN  OTHERS  THEN
     -- Assign values to variables. SQLCODE  SQLERRM cannot be used
directly.
     Err_Num = SQLCODE ;
     Err_Msg =SUBSTR( SQLERRM, 1, 100);
     -- Errors is a table existing in the database created by the  user
     INSERT INTO errors  VALUES( Err_Num, Err_Msg );
END;
  /
```

```
DECLARE
         V_Deptno NUMBER(2):= 10;
    BEGIN
         DELETE FROM dept WHERE deptno = V_Deptno ;
    EXCEPTION
         WHEN OTHERS THEN
         /* Error -2292 OCCURS WHEN REFERENTIAL INTEGRITY RULE
            IS VIOLATED */
         IF SQLCODE = - 2292  THEN
    /* Error_Log  is a table containing one varchar2 column
       created by user to store error messages */
            INSERT INTO Error_Log
          VALUES('Employees exist for Department No' || V_Deptno);
         ELSE
            INSERT INTO Error_Log
           VALUES('Unable to delete dept No' || V_Deptno ||'because
                    of  unknown  reasons');
        END IF ;
         END ;
         /
    DECLARE
      V_Dummy VARCHAR2(1);
      V_Department NUMBER(2) := 50;
    BEGIN
    SELECT 'X' INTO V_Dummy FROM dept WHERE Deptno= V_Department;
      INSERT INTO Error_Log
      VALUES ('Department ' || V_Department || 'IS ALREADY PRESENT');
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
      INSERT INTO dept
      VALUES (V_Department ,'NEW NAME', 'NEW LOC');
    END ;
    /
```

Example 6.2

7. Stored Procedures and Functions

The PL/SQL blocks in the examples discussed so far do not have any name or identity whatsoever. Even though, we have named some of the loops, we have not been able to give a name to the PL/SQL block as a whole. Hence such blocks are known as *Anonymous Blocks*.

Anonymous blocks are executed by interactively entering the block at the SQL> prompt OR by writing the PL/SQL statements in a user-named file and executing the block at SQL> prompt using @ command. The block needs to be compiled every time it is run and only the user who created the block can use the block.

Stored subprograms are named PL/SQL blocks. Stored subprograms are compiled at the time of creation and stored in the database itself. The source code is also stored in the database. Any user with necessary privileges can use the stored subprogram. Stored subprogram come in handy during application development.

Stored subprograms can be executed from the SQL> prompt using EXECUTE command. The syntax is

SQL> EXECUTE Prg_Name(Parameter_List)

Stored subprograms can also be invoked from another PL/SQL block, similar to calling a function in C.

Use SHOW ERRORS to see the Compile Time errors.

Use USER_OBJECTS, USER_ERRORS and USER_SOURCE to know about names, errors and Source code of stored subprograms.

7.1. Procedure

```
CREATE PROCEDURE  Proc_Name
     (Parameter {IN | OUT | IN OUT} datatype := value,...) AS
    Variable_Declaration ;
    Cursor_Declaration ;
    Exception_Declaration ;

BEGIN
     PL/SQL_Statements ;
EXCEPTION
     Exception_Definition ;
END Proc_Name ;
```

In a procedure declaration, it is illegal to constraint CHAR and VARCHAR2 parameters with length and NUMBER parameters with precision and scale.

If a parameter is defined as IN, then the calling program can only pass the value to the procedure. It cannot be on the left-hand side of the assignment operator.

A parameter defined as OUT is used to return a value to the caller. Any number of OUT parameters can be passed. Any OUT parameter cannot be on the right hand side of the assignment operator.

A parameter declared as IN OUT is used to pass an initial value to the procedure and on termination of the procedure, caller gets the current of value of the parameter. Any change in the value of an IN OUT parameter inside the procedure is reflected as the current value of the parameter.

IN	OUT	IN OUT
the default	must be specified	must be specified
used to pass values to the procedure.	used to return values to the caller.	Used to pass intial values to the procedure and return updated values to the caller.
formal parameter acts like a constant.	formal parameter acts like an uninitialized variable.	formal parameter acts like an initialized variable..
formal parameter cannot be assigned a value.	formal parameter cannot be used in an expression, but should be assigned a value.	formal parameter should be assigned a value.
actual parameter can be a constant, literal, initialized variable or expression.	actual parameter must be a variable.	actual parameter must be a variable.

```
CREATE OR REPLACE PROCEDURE
     Raise_Salary ( P_Empno IN NUMBER, p_raise in NUMBER) IS
     V_Cur_Salary   NUMBER ;
     Missing_Salary EXCEPTION;
BEGIN
      -- V_Cur_Salary is used only to fetch the data
     SELECT  sal into V_Cur_Salary FROM emp WHERE empno=P_Empno;
     IF  V_Cur_Salary IS NULL THEN
         RAISE  Missing_Salary;
     END IF ;
     UPDATE emp SET sal = sal + p_raise
     WHERE empno =  P_Empno ;
EXCEPTION
     WHEN NO_DATA_FOUND  THEN
        INSERT INTO Emp_Audit
        VALUES( P_Empno, 'No Employee with id ' || P_Empno);
     WHEN  Missing_Salary THEN
        INSERT INTO Emp_Audit
        VALUES( P_Empno, 'SALARY IS MISSING');
END;
/
CREATE OR REPLACE  PROCEDURE
     Get_Emp_Details( P_Empno IN NUMBER, P_Ename OUT VARCHAR2,
                      P_Sal OUT NUMBER ) IS
BEGIN
     SELECT  ename, sal into P_Ename, P_Sal  from emp
     WHERE empno=P_Empno;
EXCEPTION
     WHEN NO_DATA_FOUND THEN
         INSERT INTO Emp_Audit
         VALUES( P_Empno, 'No Employee with id ' || P_Empno);
         P_Ename := NULL;
         P_Sal := NULL;
END ;
/
```

Example 7.1

Create two bind variables Salary and Name in SQLPLUS using VARIABLE command as follows

```
SQL> VARIABLE Salary NUMBER
SQL> VARIABLE Name VARCHAR2(20)
```

Execute the procedure using the command

`EXECUTE Get_Emp_Details(7566,:Salary, :Name)`

After execution use the SQL*PLUS PRINT command to see the results

```
SQL> PRINT Salary
PRINT  Name
```

7.1.1. PARAMETER DEFAULT VALUES

Like variable declarations, the formal parameters to a procedure or function can have default values. If a parameter has default values, it does not have to be passed from the calling environment. If it is passed actual parameter will be used instead of default. Only IN parameters can have default values.

```
 PROCEDURE  Create_Dept( New_Deptno IN NUMBER ,
                     New_Dname IN  VARCHAR2  DEFAULT 'TEMP'
                     New_Loc  IN VARCHAR2  DEFAULT  'TEMP' ) IS
BEGIN
     INSERT INTO  dept
     VALUES ( New_Deptno, New_Dname, New_Loc) ;
END ;
```

Example 7.2

Now consider the following calls to Create_Dept

```
  BEGIN
      Create_Dept( 50);
      --  Actual  call will be  Create_Dept ( 50, 'TEMP', 'TEMP')
      Create_Dept ( 50, 'FINANCE');
      -- Actual call will be  Create_Dept ( 50, 'FINANCE' ,'TEMP')
      Create_Dept( 50, 'FINANCE', 'BOMBAY') ;
      -- Actual  call will be  Create_Dept(50, 'FINANCE', 'BOMBAY' )
  END;
```

```
CREATE OR REPLACE PROCEDURE Insdept
     (depno dept.deptno%TYPE,dname dept.dname%TYOE,loc dept.loc%TYPE)
     AS
     dno dept.deptno%TYPE;
BEGIN
      SELECT deptno INTO  dno FROM dept WHERE  dept.deptno=depno;
      DBMS_OUTPUT.PUT_LINE('DUPLICATE VALUE'||DNO);
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('IN NO DATA FOUND');
            INSERT INTO dept VALUES(depno,dname,loc);

END;
```

Example 7.4

```
CREATE OR REPLACE PROCEDURE findComm
(amt in out emp.sal%type)
AS
BEGIN
        SELECT sal INTO amt FROM emp WHERE comm=amt;
END;

--  Execution of the procedure findComm
DECLARE
   var1 emp.sal%type;
BEGIN
   var1 :=300;
   findComm(var1);
   DBMS_OUTPUT.PUT_LINE('Commission in exec'|| var1);
END;
```

Example 7.5

7.2. Functions

Functions are similar to procedures. They can accept one or more parameters and return a single value by using a return value. Functions can return multiple values by using OUT parameters. Functions are used as part of an expression and can be called as

```
Lvalue =Function_Name( Param1, Param2, …….)
```

Functions returning a single value for a row can be used with SQL statements.

Syntax of a FUNCTION:

```
CREATE  FUNCTION  Func_Name(Param datatype := value,..)
RETURN datatype1 AS
      Variable_Declaration ;
      Cursor_Declaration ;
      Exception_Declaration ;
BEGIN
      PL/SQL_Statements ;
      RETURN Variable_Or_Value_Of_Type_Datatype1 ;
EXCEPTION
      Exception_Definition ;
END Func_Name ;
```

Write a function, which can be used to insert a new department in dept table. Function should return the status of the insert operation.

```
CREATE FUNCTION Crt_Dept(Dno NUMBER, Dname VARCHAR2, Dloc
                         VARCHAR2)
RETURN BOOLEAN AS
BEGIN
         INSERT INTO dept
         VALUES(Dno,Dname,Dloc) ;
         RETURN TRUE ;
EXCEPTION
         WHEN OTHERS THEN
         RETURN FALSE ;
END Crt_Dept ;
/
```

Example 7.6

Function to calculate average salary of a department.

Returns Average Salary of a department.

Returns -1 if no employees are there in the department.

Returns -2 if any other error.

```
CREATE OR REPLACE FUNCTION Get_Avg_Sal(P_Deptno IN NUMBER)
RETURN NUMBER AS
     V_Sal NUMBER;
BEGIN
        SELECT Trunc(Avg(sal)) INTO V_Sal from emp
        where deptno=P_Deptno;
        IF V_Sal IS NULL THEN
           V_Sal := -1 ;
        END IF;
        RETURN V_Sal ;
EXCEPTION
        WHEN OTHERS THEN
        RETURN -2;  --Signifies any other errors
END Get_Avg_Sal ;
```

Example 7.7

Create a bind variable Avgsalary in SQLPLUS using VARIABLE command as follows

SQL> VARIABLE Avgsalary NUMBER

Execute the Function by executing the following PL/SQL block.

```
BEGIN
     :Avgsalary := Get_Avg_Sal ( 50);
END ;
/
```

After execution use the SQL*PLUS PRINT command to see the results
PRINT Avgsalary

Exceptions raised inside Procedures and Functions

If an error occurs inside a procedure, an exception (predefined or user defined) is raised. If the procedure has no exception handler for this error, control immediately passes out of the procedure to the calling environment in accordance with the propagation rules. The values of OUT and IN OUT formal parameters are not returned to actual parameters. The actual parameters will retain their old values.

7.3. Packages

Packages are PL/SQL constructs that allow related objects to be stored together.

PACKAGE consists of two parts, namely PACKAGE SPECIFICATION and PACKAGE BODY. Each of them is stored separately in Data Dictionary.

PACKAGE SPECIFICATION is used to declare functions, procedures that are part of the package. PACKAGE SPECIFICATION also contains variable and cursor declarations, which are used by the functions and procedures. Any object declared in a package specification can be referenced from other PL/SQL blocks so packages provide global variables to PL/SQL.

PACKAGE BODY contains the function and procedure definitions, which are declared in the PACKAGE SPECIFICATION. The PACKAGE BODY is optional. If the package specification does not contain any procedures or functions and contains only variable and cursor declarations then the body need not be present

All functions and procedures declared in the package specification are accessible to all users who have permissions to access the package. Users cannot access subprograms, which are defined in the package body, but not declared in the package specification. They can only be accessed by the subprograms within the package body. This facility is used to hide unwanted or sensitive information from users.

A package generally consists of functions and procedures, which are required by a specific application or a particular module of an application.

Syntax of PACKAGE SPECIFICATION:

```
CREATE PACKAGE Package_Name  AS
     Variable_Declaration ;
     Cursor_Declaration ;
     Function_Declaration ;
     Procedure_Declaration ;
END Package_Name ;
```

where

```
Function_Declaration is of the form
FUNCTION Func_Name(Param datatype,...)
RETURN datatype1 ;
Procedure_Declaration is of the form
PROCEDURE Proc_Name(Param {IN|OUT|IN OUT}datatype,...);
```

Syntax of PACKAGE BODY:

```
CREATE PACKAGE BODY Package_Name AS
     Variable_Declaration ;
      Cursor_Declaration ;
PROCEDURE Proc_Name(Param {IN|OUT|IN OUT} datatype,...} IS
BEGIN
      PL/SQL_Statements ;
END Proc_Name ;
FUNCTION Func_Name(Param datatype,...) IS
BEGIN
      PL/SQL_Statements ;
END Func_Name ;
END Package_Name ;
```

```
CREATE OR REPLACE PACKAGE Emp_Actions  AS
     --Following cursor can be used by other PL/SQl blocks
     --Procedures and Functions
     PROCEDURE  New_Employee
        ( P_Empno  NUMBER,
          P_Ename  VARCHAR2,
          P_Job    VARCHAR2,
          P_Mgr    VARCHAR2,
          P_Sal    VARCHAR2,
          P_Comm   VARCHAR2,
          P_Deptno VARCHAR2
        );
        PROCEDURE Fire_Employee( P_Empno NUMBER);
        FUNCTION  Get_Sal( P_Empno NUMBER ) RETURN NUMBER;
END  Emp_Actions ;

CREATE OR REPLACE PACKAGE BODY Emp_Actions AS
     PROCEDURE  New_Employee
          ( P_Empno NUMBER,
            P_Ename VARCHAR2,
            P_Job   VARCHAR2,
            P_Mgr   VARCHAR2,
            P_Sal   VARCHAR2,
            P_Comm  VARCHAR2,
            P_Deptno VARCHAR2
          )
AS
BEGIN
     INSERT INTO  emp VALUES( P_Empno,P_Ename,
            P_Job,P_Mgr,SYSDATE,P_Sal, P_Comm,P_Deptno);
     END  New_Employee;

PROCEDURE Fire_Employee ( P_Empno NUMBER) IS
BEGIN
     DELETE FROM emp  WHERE empno=P_Empno ;
     END  Fire_Employee ;

FUNCTION Get_Sal ( P_Empno NUMBER)
RETURN NUMBER AS
     V_Sal NUMBER ;
BEGIN
     SELECT sal INTO V_Sal FROM emp WHERE empno =P_Empno;
     RETURN V_Sal;
EXCEPTION
     WHEN NO_DATA_FOUND THEN
     RAISE_APPLICATION_ERROR( -20999 , 'Employee Not Present');
     END  Get_Sal ;
END Emp_Actions;
```

Example 7.9

The procedure and functions calls are the same as in standalone subprograms. The packaged procedures and functions have to be prefixed with package names.

```
BEGIN
     Emp_Actions.New_Employee( 10, 'ABCD', …….);
     V_Sal := Emp_Actions. Get_Sal( 7566);
END ;
```

The first time a package is called, it is instantiated. This means that the package is read from disk into memory and P-CODE is run. At this point memory is allocated for any variables defined in the package. Each session will have its own copy of packaged variables, so there is no problem of two simultaneous sessions accessing the same memory locations.

8. Table of Examples

9. Appendix A

Guidelines for Good SQL:

- SQL statements should be reviewed for performance to determine if they are being executed in the most optimal manner. The following standards are performance guidelines for SQL statements to ensure that the SQL statements are performing the most efficient way possible.

- Avoid using arithmetic expressions or functions on the left hand side of the SELECT criteria (as in WHERE clause) that involves a column name; since it will not use index.

- '[NOT] EXISTS' should be used rather than '[NOT] IN' when using a subquery that tests existence or absence of a row.

- When a query is having an 'OR' in a 'WHERE' clause, and both of the criteria sections of the 'WHERE/OR' clause are specifying different indexes, and one of the indices is desired over the other, use a function on the portion of the 'OR' that is not desired to force an index.

- For example, in 'WHERE PART_NO = '3038M83P09' OR CUSTOMER_NO = '123456'', if both of these force different indices, and it is desired to have the CUSTOMER_NO index used, then use some function like 'WHERE NVL(PART_NO,0)='3030M83P09' OR ....'.

- To use an index, the columns in the WHERE clause should be ordered in the order they appear on the desired index.

- When using more than one table on a FROM clause, the table that is having the larger number of rows should be listed first.

- When using an '... IN' clause, the elements in the subset (,,,) should be listed in alphabetical order.

- When poor performance exists with very complex joins, the UNION clause should be used instead of a normal join, whenever possible.

- For numeric columns that allow nulls, NVL function should be used

10. Appendix B

Dept Table

Name	Type
DEPTNO	NUMBER(2)
DNAME	VARCHAR2(14)
LOC	VARCHAR2(13)

Deptno	Dname	Loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Emp Table

Name	Type
EMPNO	NUMBER(4)
ENAME	VARCHAR2(10)
JOB	VARCHAR2(9)
MGR	NUMBER(4)
HIREDATE	DATE
SAL	NUMBER(7,2)
COMM	NUMBER(7,2)
DEPTNO	NUMBER(2)

Empno	Ename	Job	Mgr	Hiredate	Sal	Comm	Dept
7369	SMITH	CLERK	7902	17-Dec-80	800		20
7499	ALLEN	SALESMAN	7698	20-Feb-81	1600	300	30
7521	WARD	SALESMAN	7698	22-Feb-81	1250	500	30
7566	JONES	MANAGER	7839	2-Apr-81	2975		20
7654	MARTIN	SALESMAN	7698	28-Sep-81	1250	1400	30
7698	BLAKE	MANAGER	7839	1-May-81	2850		30
7782	CLARK	MANAGER	7839	9-Jun-81	2450		10
7788	SCOTT	ANALYST	7566	9-Dec-82	3000		20
7839	KING	PRESIDENT		17-Nov-81	5000		10
7844	TURNER	SALESMAN	7698	8-Sep-81	1500	0	30
7876	ADAMS	CLERK	7788	12-Jan-83	1100		20
7900	JAMES	CLERK	7698	3-Dec-81	950		30
7902	FORD	ANALYST	7566	3-Dec-81	3000		20
7934	MILLER	CLERK	7782	23-Jan-82	1300		10